REST

Construa API's inteligentes de maneira simples

ALEXANDRE SAUDATE

Sumário

Versão: 16.3.8

Capítulo 1

Por que utilizar REST?

"Não é merecedor do favo de mel aquele que evita a colméia porque as abelhas têm ferrões"
– William Shakespeare

Você é um empreendedor. Seguindo a filosofia de uma *startup*, você quer produzir um *site* para seu produto, uma loja de cervejas finas pela internet. Assim, você quer fazer um sistema que seja rápido de produzir e, ao mesmo tempo, seja eficiente e fácil de reutilizar em vários ambientes diferentes (você está considerando uma versão *mobile* para o sistema). Assim, você explora diversas possibilidades.

1.1 HTML tradicional versus REST

Você decide avaliar, antes de tudo, sistemas tradicionais web, baseados em formulários HTML.

Fica claro para você que fazer um sistema tradicional, baseado em formulários HTML, está fora de questão. Isso porque esse tipo de formulário é claramente oti-

mizado para trabalhar com sistemas baseados na web, mas você não tem certeza em relação ao quanto ele será flexível para trabalhar com outros tipos de *front-end*. Por exemplo, considere uma tela para cadastro dos clientes do *site*:

```
<html>
   <body>
      <form action="/cadastrar" method="post">
         <input type="text" name="nome" />
         <input type="text" name="dataNascimento" />
         <input type="submit" value="Cadastrar" />
      </form>
   </body>
</html>
```

Este código (com um pouco mais de tratamento) deve te atender bem. No entanto, após submeter o formulário, o servidor deve te trazer uma página HTML - o que, obviamente, não é desejável num cenário com vários *front-ends*, como um aplicativo para Android ou iOS.

Sua próxima possibilidade é, portanto, avaliar outros tipos de sistemas.

1.2 Web services

O próximo cenário que vem à sua mente é o de uso de *web services* tradicionais, ou seja, baseados em SOAP (*Simple Object Access Protocol*). A princípio, te parece uma boa idéia, já que é o mecanismo de tráfego de informações é via XML e, portanto, é possível interagir com serviços desse tipo a partir de código *javascript* (ou seja, possibilitando a interação tanto a partir de um *browser* quanto a partir de código puro). Porém, estudando mais você percebe que interagir com este tipo de *web service* não é tão simples quanto você gostaria. Por exemplo, para listar os clientes do seu sistema, é necessário enviar o seguinte XML para o servidor:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <listarClientes xmlns="http://brejaonline.com.br/administracao/1.0/service"
   </soap:Body>
</soap:Envelope>
```

Ao que você receberá como resposta:

```
<soap:Envelope xmlns:domain="http://brejaonline.com.br/administracao/1.0/domain"
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

```
    <soap:Body>
        <listarClientesResponse xmlns="http://brejaonline.com.br/administracao/1.0/
            <domain:clientes>
                <domain:cliente domain:id="1">
                    <domain:nome>Alexandre</domain:nome>
                    <domain:dataNascimento>2012-12-01</domain:dataNascimento>
                </domain:cliente>
                <domain:cliente domain:id="2">
                    <domain:nome>Paulo</domain:nome>
                    <domain:dataNascimento>2012-11-01</domain:dataNascimento>
                </domain:cliente>
            </domain:clientes>
        </listarClientesResponse>
    </soap:Body>
</soap:Envelope>
```

Este código, então, te parece complicado demais para uma simples requisição de listagem de clientes. A complexidade (e verbosidade) do protocolo fica especialmente evidente quando você pensa no caso da comunicação feita com plataformas móveis, onde a comunicação pela rede deve ser a mais sucinta possível.

1.3 REST

Você estuda mais a respeito de *web services* quando você lê sobre serviços REST. Ao estudar sobre o assunto, te parece uma técnica simples. Por exemplo, para realizar a listagem de clientes do seu sistema, você pode utilizar o navegador (seja este o `Chrome`, `Firefox`, `Internet Explorer` ou qualquer outro) para abrir a URL http://localhost:8080/cervejaria/clientes . Ao fazer isso, você pode obter o seguinte resultado:

```
<clientes>
    <cliente id="1">
        <nome>Alexandre</nome>
        <dataNascimento>2012-12-01</dataNascimento>
    </cliente>
    <cliente id="2">
        <nome>Paulo</nome>
        <dataNascimento>2012-11-01</dataNascimento>
    </cliente>
</clientes>
```

Mas, afinal, como fazer seu código interagir com esta listagem? O que aconteceu?

Os princípios básicos de REST

REST significa *REpresentational State Transfer* (ou `Transferência de Estado Representativo`, em tradução livre), e é um estilo de desenvolvimento de *web services* que teve origem na tese de doutorado de Roy Fielding. Este, por sua vez, é co-autor de um dos protocolos mais utilizados no mundo, o HTTP (`HyperText Transfer Protocol`). Assim, é notável que o protocolo REST é guiado (dentre outros preceitos) pelo que seriam as boas práticas de uso de HTTP:

- Uso adequado dos métodos HTTP;
- Uso adequado de URL's;
- Uso de códigos de status padronizados para representação de sucessos ou falhas;
- Uso adequado de cabeçalhos HTTP;
- Interligações entre vários recursos diferentes.

O propósito deste livro é, portanto, exemplificar o que são estas práticas e guiar você, leitor, através das mesmas e fornecer os insumos de como construir seus próprios serviços, de maneira que estes sejam sempre tão escaláveis, reutilizáveis e manuteníveis quanto possível.

As fundações de REST

O "marco zero" de REST é o **recurso**. Em REST, tudo é definido em termos de recursos, sendo estes os conjuntos de dados que são trafegados pelo protocolo. Os recursos são representados por URI's. Note que, na *web*, URI's e URL's são essencialmente a mesma coisa - razão pela qual vou usar os dois termos neste livro de forma intercalada.

Qual a diferença entre uma URL e uma URI?

URL significa `Universal Resource Locator` e URI, `Universal Resource Identifier`. Uma URI, como diz o próprio nome, pode ser utilizada para identificar qualquer coisa - dar um caminho para um determinado conteúdo, dar nome a este, etc. Já uma URL pode ser utilizada apenas para fornecer caminhos - sendo que uma URL é, portanto, uma forma de uma URI. É mais natural que URI's que não sejam URL's sejam utilizadas em outros contextos, como fornecimento de *namespaces* XML.

Tomando como exemplo o caso da listagem de clientes, é possível decompôr a URL utilizada para localização da listagem em várias partes:

http://localhost:8080/cervejaria/clientes

- **http://** - Indica o protocolo que está sendo utilizado (no caso, HTTP);
- **localhost:8080** - Indica o servidor de rede que está sendo utilizado e a porta (quando a porta não é especificada, assume-se que é a padrão - no caso do protocolo HTTP, 80);
- **cervejaria** - Indica o **contexto** da aplicação, ou seja, a raiz pela qual a aplicação está sendo fornecida para o cliente. Vou me referir a esta, daqui em diante, como **contexto da aplicação** ou apenas **contexto**;
- **clientes** - É o endereço, de fato, do recurso - no caso, a listagem de clientes. Vou me referir a este, daqui em diante, como **endereço do recurso**.

O protocolo

O protocolo, em realidade, não é uma restrição em REST - em teoria. Na prática, o HTTP é o único protocolo 100% compatível conhecido. Vale destacar que o HTTPS (`HyperText Transfer Protocol over Secure Sockets Layer`) não é uma variação do HTTP, mas apenas a adição de uma camada extra - o que mantém o HTTPS na mesma categoria que o HTTP, assim como qualquer outro protocolo que seja utilizado sobre o HTTP, como SPDY.

As possíveis causas para esta "preferência" podem ser apontadas: o autor do HTTP também é o autor de REST e, além disso, o protocolo HTTP é um dos mais

utilizados no mundo (sendo que a *web*, de maneira geral, é fornecida por este protocolo). Estes fatos promoveriam uma aceitação grande e rápida absorção de REST pela comunidade de desenvolvedores.

A URL

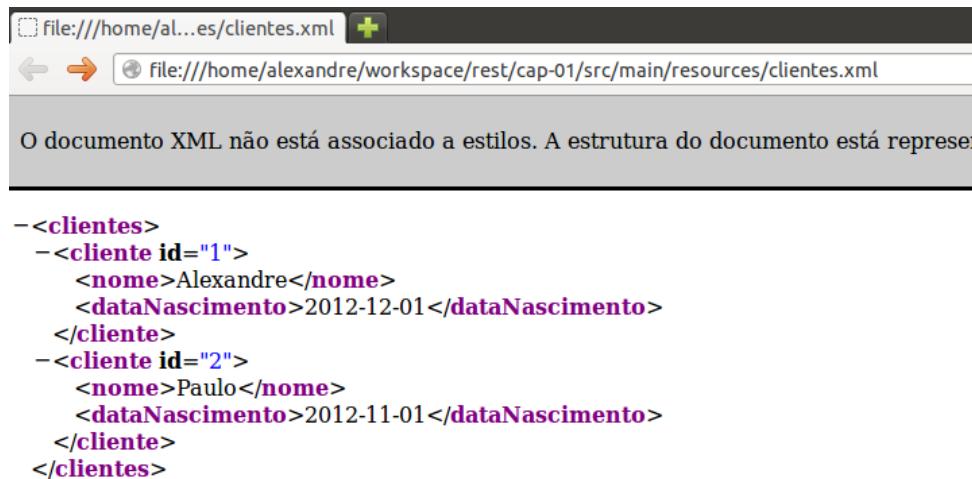
A URL escolhida deve ser única por recurso. Isto significa que, sempre que desejar obter a representação de todos os clientes, você deve utilizar a URL http://localhost:8080/cervejaria/clientes . Realizar uma consulta nesta URL pode retornar dados da seguinte maneira:

```
<clientes>
   <cliente id="1">
      <nome>Alexandre</nome>
      <dataNascimento>2012-12-01</dataNascimento>
   </cliente>
   <cliente id="2">
      <nome>Paulo</nome>
      <dataNascimento>2012-11-01</dataNascimento>
   </cliente>
</clientes>
```

Se você desejar, portanto, retornar um cliente específico, você deve utilizar uma URL diferente. Suponha, por exemplo, que você deseja retornar o cliente com `id` igual a 1. Neste caso, a URL seria http://localhost:8080/cervejaria/clientes/1 . Isso retornaria algo como:

```
<cliente id="1">
   <nome>Alexandre</nome>
   <dataNascimento>2012-12-01</dataNascimento>
</cliente>
```

1.4 Desenvolvendo um protótipo de web service REST

Para desenvolver um protótipo de *web service* REST bem simples, não é necessário utilizar-se de alta tecnologia. Basta que o desenvolvedor tenha em mente o resultado que deseja alcançar e criar um arquivo contendo a resposta desejada. Por exemplo, suponha um arquivo `clientes.xml`, com o conteúdo:

```
<clientes>
   <cliente id="1">
      <nome>Alexandre</nome>
      <dataNascimento>2012-12-01</dataNascimento>
   </cliente>
   <cliente id="2">
      <nome>Paulo</nome>
      <dataNascimento>2012-11-01</dataNascimento>
   </cliente>
</clientes>
```

Para acessá-lo pelo *browser*, basta utilizar a URL `file://<caminho do arquivo>`. Por exemplo, no meu caso (ambiente Linux), o arquivo está em `/home/alexandre/workspace/rest/cap-01/src/main/resources/clientes.xml`. Assim, a URL completa fica file:///home/alexandre/workspace/rest/cap-01/src/main/resources/clientes.xml - note que são três barras no começo (duas para delimitação do protocolo e outra para indicar a raiz do sistema Linux).

URL para arquivos no Windows

No Windows, para renderizar uma URL também é necessário utilizar três barras no começo. Por exemplo, para localizar um arquivo em `Documentos`, o caminho a ser inserido no *browser* ficaria similar a file:///C:/Users/Alexandre/Documents/clientes.xml . Isto é devido a uma particularidade do próprio sistema de arquivos do Windows, que pode ter várias raízes (C:, D:, e assim por diante).

Desta forma, o conteúdo é renderizado pelo *browser*:

file:///home/al...es/clientes.xml

file:///home/alexandre/workspace/rest/cap-01/src/main/resources/clientes.xml

O documento XML não está associado a estilos. A estrutura do documento está represen

```
-<clientes>
  -<cliente id="1">
      <nome>Alexandre</nome>
      <dataNascimento>2012-12-01</dataNascimento>
    </cliente>
  -<cliente id="2">
      <nome>Paulo</nome>
      <dataNascimento>2012-11-01</dataNascimento>
    </cliente>
  </clientes>
```

Figura 1.1: XML renderizado pelo browser

Para desenvolver um cliente para este arquivo, basta utilizar a API de I/O do próprio Java, começando pela classe `java.net.URL`, que indica qual o caminho a ser seguido:

```
String caminho = "file:///home/alexandre/workspace/rest/" +
        "cap-01/src/main/resources/clientes.xml";
URL url = new URL(caminho);
```

Feito isso, o próximo passo é conectar-se a este caminho e abrir uma *input stream*, ou seja, um canal de leitura de dados do caminho indicado pela URL. Para conectar-se, basta utilizar o método `openConnection` da URL, e para abrir a *input stream*, basta utilizar o método `getInputStream`:

```
java.io.InputStream inputStream = url.openConnection().getInputStream();
```

O próximo passo é encapsular esta leitura em um `java.io.BufferedReader`. Esta classe possui um método utilitário para ler informações linha a linha, facilitando o processo de leitura de dados. Para utilizá-lo, no entanto, é necessário encapsular a `InputStream` em um `java.io.InputStreamReader`:

```
BufferedReader bufferedReader = new BufferedReader(
                new InputStreamReader(inputStream));
```

Uma vez feito esse procedimento, basta utilizar o método `readLine` em um laço, até que este método retorne `null`:

```
String line = null;

while ((line = bufferedReader.readLine()) != null) {
   System.out.println(line);
}
```

O código completo fica assim:

```
package br.com.cervejaria.cliente;

import java.io.*;
import java.net.URL;

public class Cliente {

   public static void main(String[] args) throws IOException {
       String caminho = "file:///home/alexandre/workspace/rest/"
               + "cap-01/src/main/resources/clientes.xml";
        URL url = new URL(caminho);
        InputStream inputStream = url.openConnection().getInputStream();
        BufferedReader bufferedReader = new BufferedReader(
                new InputStreamReader(inputStream));

        String line = null;

        while ((line = bufferedReader.readLine()) != null) {
            System.out.println(line);
        }
    }
}
```

1.5 Avançando o protótipo de web service REST

Para desenvolver um protótipo que seja verdadeiramente baseado na web, basta um simples *servlet*. No código-fonte deste livro, estou utilizando a especificação 3.0 de *servlets*, que me permite anotá-los para que sejam detectados pelo contêiner.

Para realizar a listagem de clientes, basta fazer com que o *servlet* leia o arquivo `clientes.xml`, utilizando anteriormente, Assim, o *servlet* será capaz de fornecer

estas informações para o cliente. Este fornecimento de dados vai ser feito via método `GET` (mais à frente, no capítulo X, você vai entender o motivo). O código do *servlet* vai ficar assim:

```
package br.com.cervejaria.servlet;

import java.io.*;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet(urlPatterns = "/clientes", loadOnStartup = 1)
public class ClientesServlet extends HttpServlet {

   private String clientes;

   @Override
   protected void doGet(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
      resp.getWriter().print(clientes);
   }

   @Override
   public void init() throws ServletException {
      // Faz aqui a leitura do arquivo clientes.xml
      // E coloca na String clientes
   }

}
```

O código-fonte do livro

No código-fonte do livro (disponível em https://github.com/alesaudate/rest/) , você encontra um projeto Maven com este código pronto para ser inicializado. Basta executar `mvn jetty:run` e acessar esta URL.

Ao acessar a URL http://localhost:8080/cervejaria/clientes, você obtém como

resposta exatamente o conteúdo do arquivo. E para realizar isto com código, basta alterar o código fonte do cliente criado anteriormente para esta mesma URL:

```
URL url = new URL("http://localhost:8080/cervejaria/clientes");
```

1.6 Sumário

Neste capítulo, você pode conferir alguns dos principais motivos pelos quais o modelo REST é selecionado para se trabalhar em aplicações modernas. Você conferiu alguns dos princípios mais básicos de REST, como o uso de URL's e como realizar o acesso a estes documentos através de *browsers* e código Java.

Resta, ainda, conferir os princípios do protocolo HTTP (nos quais REST é fortemente baseado) e entender como criar serviços mais complexos utilizando este paradigma.

Capítulo 2

O protocolo HTTP

"Se vi mais longe, foi por estar de pé sobre ombros de gigantes"
– Isaac Newton

Como mencionado anteriormente, o modelo REST foi desenvolvido por Roy Fielding, um dos criadores do protocolo HTTP. Portanto, fica naturalmente evidente, para quem conhece ambos os modelos, as semelhanças entre ambos - sendo que, na realidade, REST é idealmente concebido para uso com o protocolo HTTP. Portanto, para ser um bom usuário de REST, é necessário ter um bom conhecimento do protocolo HTTP.

2.1 Os fundamentos do HTTP

O protocolo HTTP (*HyperText Transfer Protocol* - Protocolo de Transferência de Hipertexto) data de 1996, época em que os trabalhos conjuntos de Tim Berners-Lee, Roy Fielding e Henrik Frystyk Nielsen levaram à publicação de uma RFC (*Request for Comments*) descrevendo este protocolo. Trata-se de um protocolo de camada de

aplicação (segundo o modelo OSI) e, portanto, de relativa facilidade de manipulação em aplicações.

Este protocolo foi desenvolvido de maneira a ser o mais flexível possível para comportar diversas necessidades diferentes. Em linhas gerais, este protocolo segue o seguinte formato de requisições:

```
<método> <URL> HTTP/<versão>
<Cabeçalhos - Sempre vários, um em cada linha>

<corpo da requisição>
```

Por exemplo, para realizar a requisição do capítulo passado, algo semelhante ao seguinte foi enviado para o servidor:

```
GET /cervejaria/clientes HTTP/1.1
Host: localhost:8080
Accept: text/html
```

Portanto, o que foi passado nesta requisição foi:

```
GET /cervejaria/clientes HTTP/1.1
```

Com isso, o método `GET` foi utilizado para solicitar o conteúdo da URL `/cervejaria/clientes`. Além disso, o protocolo HTTP versão 1.1 foi utilizado.

Note que o *host*, ou seja, o servidor responsável por fornecer estes dados, é passado em um **cabeçalho** à parte. No caso, o *host* escolhido foi `localhost`, na porta `8080`.

Além disso, um outro cabeçalho, `Accept`, foi fornecido.

A resposta para as requisições seguem o seguinte formato geral:

```
HTTP/<versão> <código de status> <descrição do código>
<cabeçalhos>

<resposta>
```

Por exemplo, a resposta para esta requisição foi semelhante à seguinte:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 245
```

```
<clientes>
   <cliente id="1">
      <nome>Alexandre</nome>
      <dataNascimento>2012-12-01</dataNascimento>
   </cliente>
   <cliente id="2">
      <nome>Paulo</nome>
      <dataNascimento>2012-11-01</dataNascimento>
   </cliente>
</clientes>
```

Aqui, note que o código 200 indicou que a requisição foi bem-sucedida, e o cabeçalho `Content-Length` trouxe o tamanho da resposta - no caso, o XML que contém a listagem de clientes. Além disso, o cabeçalho `Content-Type` indica que a resposta é, de fato, XML - através de um tipo conhecido como `Media Type`.

2.2 Métodos HTTP

A versão corrente do HTTP, 1.1, define oficialmente oito métodos - embora o protocolo seja extensível em relação a estes métodos. Hoje, estes oito são:

- GET
- POST
- PUT
- DELETE
- OPTIONS
- HEAD
- TRACE
- CONNECT

Cada método possui particularidades e aplicações de acordo com a necessidade. Estas particularidades são definidas em termos de **idempotência**, **segurança** e **mecanismo de passagem de parâmetros**. Além disso, por cada um possuir particularidades de uso, considerarei neste livro apenas os seis primeiros - os métodos `TRACE` e `CONNECT` serão considerados fora do escopo deste livro.

Idempotência

A idempotência de um método é relativa às modificações que são realizadas em informações do lado do servidor. Trata-se do efeito que uma mesma requisição tem do lado do servidor - se a mesma requisição, realizada múltiplas vezes, provoca alterações no lado do servidor como se fosse uma única, então esta é considerada idempotente.

Por exemplo, considere as quatro operações de bancos de dados: `SELECT`, `INSERT`, `UPDATE` e `DELETE`. Realizando um paralelo destas com o conceito de idempotência, observe o seguinte:

```
SELECT * from CLIENTES;
```

Note que esta requisição, para um banco de dados, terá o mesmo efeito todas as vezes em que for executada (obviamente, assumindo que ninguém está fazendo alterações nas informações que já estavam gravadas).

Agora, observe o seguinte:

```
INSERT INTO CLIENTES VALUES (1,'Alexandre');
```

Esta requisição, por sua vez, provocará diferentes efeitos sobre os dados do banco de dados todas as vezes que for executada, dado que está aumentando o tamanho da tabela. Portanto, ela não é considerada idempotente.

Note que este conceito não está relacionado à realização ou não de modificações. Por exemplo, considere as operações `UPDATE` e `DELETE`:

```
UPDATE CLIENTES SET NOME = 'Paulo' WHERE ID = 1;
```

```
DELETE FROM CLIENTES WHERE ID = 1;
```

Note que, em ambos os casos, as alterações realizadas são idênticas à todas as subsequentes. Por exemplo, suponha os dados:

```
ID NOME
---------
1  Alexandre
```

Se a requisição de atualização for enviada, estes dados ficarão assim:

```
ID NOME
---------
1  Paulo
```

E então, caso a mesma requisição seja enviada repetidas vezes, ainda assim provocará o mesmo efeito que da primeira vez, sendo considerada, portanto, idempotente.

Segurança

Quanto à **segurança**, os métodos são assim considerados se não provocarem quaisquer alterações nos dados contidos. Ainda considerando o exemplo das operações de bancos de dados, por exemplo, o método `SELECT` pode ser considerado seguro - `INSERT`, `UPDATE` e `DELETE`, não.

Em relação a estas duas características, a seguinte distribuição dos métodos HTTP é feita:

	Idempotente	Seguro
GET	X	X
POST		
PUT	X	
DELETE	X	
HEAD	X	X
OPTIONS	X	X

Figura 2.1: Métodos HTTP, de acordo com idempotência e segurança

2.3 Tipos de passagem de parâmetros

Os métodos HTTP suportam parâmetros sob duas formas: os chamados *query parameters* e *body parameters*.

Os *query parameters* são passados na própria URL da requisição. Por exemplo, considere a seguinte requisição para o serviço de busca do Google:

```
http://www.google.com.br/?q=HTTP
```

Esta requisição faz com que o servidor do Google automaticamente entenda a string `HTTP` como um parâmetro. Inserir esta URL no *browser* automaticamente indica para o servidor que uma busca por HTTP está sendo realizada.

Os *query parameters* são inseridos a partir do sinal de interrogação. Este sinal indica para o protocolo HTTP que, de alí em diante, serão utilizados *query parameters*. Esses são inseridos com formato <chave>=<valor> (assim como no exemplo,

em que `q` é a chave e `HTTP`, o valor). Caso mais de um parâmetro seja necessário, os pares são separados com um `&`. Por exemplo, a requisição para o serviço do Google poderia, também, ser enviada da seguinte maneira:

```
http://www.google.com.br/?q=HTTP&oq=HTTP
```

Os *query parameters* são enviados na própria URL. Isto quer dizer que a requisição para o Google é semelhante à seguinte:

```
GET /?q=HTTP&oq=HTTP HTTP/1.1
Host: www.google.com.br
```

Note que o caracter de espaço é o separador entre o método HTTP, a URL e a versão do HTTP a ser utilizada. Desta forma, se for necessário utilizar espaços nos *query parameters*, é necessário utilizar uma técnica chamada de **codificação da URL**. Esta técnica adapta as URL's para que elas sejam compatíveis com o mecanismo de envio de dados, e codificam não apenas espaços como outros caracteres especiais e caracteres acentuados.

Esta codificação segue a seguinte regra:

- Espaços podem ser codificados utilizando + ou a string **%20**
- Letras maiúsculas e minúsculas, números e os caracteres **.**, -, ~ e _ são deixados como estão
- Caracteres restantes são codificados de acordo com sua representação ASCII / UTF-8 e codificados como hexadecimal

Assim, a *string* **às vezes** é codificada como **%C3%A0s+vezes** (sendo que os *bytes* C3 e A0 representam os números 195 e 160 que, em codificação UTF-8, tornam-se a letra **à**).

Note que uma limitação dos *query parameters* é a impossibilidade de passar dados estruturados como parâmetro. Por exemplo, não há a capacidade de relacionar um *query param* com outro, levando o desenvolvedor a criar maneiras de contornar esta limitação. Por exemplo, suponha que seja necessário desenvolver uma pesquisa de clientes baseada em vários tipos de impostos devidos num certo período de tempo. Esta pesquisa deveria fornecer:

- O nome do imposto

- O período (data inicial e data final da busca)

Se esta pesquisa for baseada em um único imposto, não há problema algum. Mas tome como base uma pesquisa baseada como uma lista de impostos. Ela seria semelhante a:

```
/pessoas?imposto.1=IR&data.inicio.1=2011-01-01&data.inicio.2=2012-01-01
```

Onde a numeração, neste caso, seria o elemento agrupador dos dados - ou seja, um *workaround* para a limitação de dados estruturados.

Quando há a necessidade de fornecer dados complexos, no entanto, é possível fazê-lo pelo corpo da requisição. Algo como:

```
POST /clientes HTTP/1.1
Host: localhost:8080
Content-Type: text/xml
Content-Length: 93

<cliente>
   <nome>Alexandre</nome>
   <dataNascimento>2012-01-01</dataNascimento>
</cliente>
```

Quanto a este tipo de parâmetro, não há limitações. O servidor faz a interpretação dos dados a partir do fato de que esses parâmetros são os últimos do documento, e controla o tamanho da leitura utilizando o cabeçalho `Content-Length`.

2.4 Cabeçalhos

Os cabeçalhos, em HTTP, são utilizados para trafegar todo o tipo de meta informação a respeito das requisições. Vários destes cabeçalhos são padronizados; no entanto, eles são facilmente extensíveis para comportar qualquer particularidade que uma aplicação possa requerer nesse sentido.

Por exemplo, ao realizar uma requisição, pelo Firefox, para o site http://www.casadocodigo.com.br/, as seguintes informações são enviadas:

```
GET / HTTP/1.1
Host: www.casadocodigo.com.br
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:19.0) Gecko/20100101 Fire
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Como checar o tráfego via HTTP

Para visualizar o tráfego destes dados, várias técnicas estão à disposição. As mais simples são a instalação de um *plugin* do Firefox chamado `Live HTTP Headers` ou apertar `F12` no Chrome.

Portanto, os seguintes cabeçalhos são enviados: `Host`, `User-Agent`, `Accept`, `Accept-Language`, `Accept-Encoding` e `Connection`. Cada um desses tem um significado para o servidor - no entanto, o protocolo HTTP não exige nenhum deles. Ao estabelecer esta liberdade, tanto o cliente quanto o servidor estão livres para negociar o conteúdo da maneira como acharem melhor.

Obviamente, isto não quer dizer que estes cabeçalhos não são padronizados. Da lista acima, cada cabeçalho possui uma explicação:

- `Host` - mostra qual foi o DNS utilizado para chegar a este servidor
- `User-Agent` - fornece informações sobre o meio utilizado para acessar este endereço
- `Accept` - realiza negociação com o servidor a respeito do conteúdo aceito (mais informações na seção 2.4).
- `Accept-Language` - negocia com o servidor qual o idioma a ser utilizado na resposta
- `Accept-Encoding` - negocia com o servidor qual a codificação a ser utilizada na resposta
- `Connection` - ajusta o tipo de conexão com o servidor (persistente ou não).

Tecnicamente falando, os cabeçalhos são utilizados para tráfego de *meta dados* - ou seja, informações a respeito da informação "de verdade". Tome um cenário de envio de *emails*: o destinatário, o assunto e outras informações são apenas *meta dados* - a verdadeira informação é o conteúdo do *email*. Desta forma, uma requisição para um serviço HTTP de envio de *emails* poderia ser feita da seguinte forma:

```
POST /email HTTP/1.1
Host: brejaonline.com.br
Destinatario: alesaudate@gmail.com
Assunto: Quanto custa a cerveja dos Simpsons?
```

Vários usos diferentes são dados para cabeçalhos em HTTP. Ao longo deste livro, vou mencioná-los quando necessário.

2.5 Media Types

Ao realizar uma requisição para o site http://www.casadocodigo.com.br , o `Media Type text/html` é utilizado, indicando para o navegador qual é o tipo da informação que está sendo trafegada (no caso, HTML). Isto é utilizado para que o cliente saiba como trabalhar com o resultado (e não com tentativa e erro, por exemplo), dessa maneira, os `Media Types` são formas padronizadas de descrever uma determinada informação.

Os `Media Types` são divididos em *tipos* e *subtipos*, e acrescidos de parâmetros (se houverem). São compostos com o seguinte formato: **tipo/subtipo**. Se houverem parâmetros, o **;** (ponto-e-vírgula) será utilizado para delimitar a área dos parâmetros. Portanto, um exemplo de `Media Type` seria **text/xml; charset="utf-8"** (para descrever um XML cuja codificação seja UTF-8).

Os tipos mais comuns são:

- **application**
- **audio**
- **image**
- **text**
- **video**
- **vnd**

Cada um desses tipos é utilizado com diferentes propósitos. **Application** é utilizado para tráfego de dados específicos de certas aplicações. **Audio** é utilizado para formatos de áudio. **Image** é utilizado para formatos de imagens. **Text** é utilizado para formatos de texto padronizados ou facilmente inteligíveis por humanos. **Video**

é utilizado para formatos de vídeo. **Vnd** é para tráfego de informações de softwares específicos (por exemplo, o Microsoft Office).

Em serviços REST, vários tipos diferentes de `Media Types` são utilizados. As maneiras mais comuns de representar dados estruturados, em serviços REST, são via XML e JSON, que são representados pelos `Media Types` `application/xml` e `application/json`, respectivamente. O XML também pode ser representado por **text/xml**, desde que possa ser considerado legível por humanos.

Note que subtipos mais complexos do que isso podem ser representados com o sinal + (mais). Este sinal é utilizado em vários subtipos para delimitação de mais de um subtipo. Este é o caso com XHTML, por exemplo, que denota o tipo HTML acrescido das regras de XML, e cujo `media type` é `application/xhtml+xml`. Outro caso comum é o do protocolo SOAP, cujo `media type` é `application/soap+xml`.

Os `Media Types` são negociados a partir dos cabeçalhos `Accept` e `Content-Type`. O primeiro é utilizado em requisições, e o segundo, em respostas.

Ao utilizar o cabeçalho `Accept`, o cliente informa ao servidor qual tipo de dados espera receber. Caso seja o tipo de dados que não possa ser fornecido pelo servidor, o mesmo retorna o código de erro `415` (apresentado na seção 2.6), indicando que o *Media Type* não é suportado. Se o tipo de dados existir, então os dados são fornecidos e o cabeçalho `Content-Type` apresenta qual é o tipo de informação fornecida.

Existem várias formas de realizar esta solicitação. Caso o cliente esteja disposto a receber mais de um tipo de dados, o cabeçalho `Accept` pode ser utilizado para esta negociação. Por exemplo, se o cliente puder receber qualquer tipo de imagem, esta solicitação pode utilizar um **curinga**, representado por * (asterisco). O servidor irá converter esta solicitação em um tipo adequado. Esta negociação será similar à seguinte:

Requisição:

```
GET /foto/1 HTTP/1.1
Host: brejaonline.com.br
Accept: image/*
```

Resposta:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 10248
```

Além disso, também é possível solicitar mais de um tipo de dados apenas separando-os por **,** (vírgula). Por exemplo, um cliente de uma página web poderia realizar uma solicitação como a seguinte:

```
GET / HTTP/1.1
Host: brejaonline.com.br
Accept:text/html,*/*
```

Note, assim, que o cliente solicita uma página HTML que se uma não estiver disponível, qualquer tipo de conteúdo é aceito - como pode ser observado pela presença de dois curingas, em tipo e subtipo de dados. Neste caso, a prioridade atendidada é a da **especificidade**, ou seja, como **text/html** é mais específico, tem mais prioridade.

Em caso de vários tipos de dados que não atendam a esta regra, o parâmetro **q** pode ser adotado para que se possa realizar diferenciações de peso. Este parâmetro recebe valores variando entre 0.1 e 1, contendo a ordem de prioridade. Por exemplo, suponha a seguinte requisição:

```
GET / HTTP/1.1
Host: brejaonline.com.br
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

Isto significa que tanto HTML quanto XHTML têm prioridade máxima. Na falta de uma representação do recurso com qualquer um desses tipos de dados, o XML é aceito com prioridade 90%. Na ausência do XML, qualquer outra representação é aceita, com prioridade 80%.

2.6 Códigos de status

Toda requisição que é enviada para o servidor retorna um código de status. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, sendo:

- 1xx - Informacionais
- 2xx - Códigos de sucesso
- 3xx - Códigos de redirecionamento
- 4xx - Erros causados pelo cliente
- 5xx - Erros originados no servidor

De acordo com Leonard Richardson e Sam Ruby, os códigos mais importantes e mais utilizados são:

2xx

200 - OK

Indica que a operação indicada teve sucesso.

201 - Created

Indica que o recurso desejado foi criado com sucesso. Deve retornar um cabeçalho `Location`, que deve conter a URL onde o recurso recém-criado está disponível.

202 - Accepted

Indica que a solicitação foi recebida e será processada em outro momento. É tipicamente utilizada em requisições assíncronas, que não serão processadas em tempo real. Por esse motivo, pode retornar um cabeçalho `Location`, que trará uma URL onde o cliente pode consultar se o recurso já está disponível ou não.

204 - No Content

Usualmente enviado em resposta a uma requisição `PUT`, `POST` ou `DELETE`, onde o servidor pode recusar-se a enviar conteúdo.

206 - Partial Content

Utilizado em requisições `GET` parciais, ou seja, que demandam apenas parte do conteúdo armazenado no servidor (caso muito utilizado em servidores de *download*).

3xx

301 - Moved Permanently

Significa que o recurso solicitado foi realocado permanentemente. Uma resposta com o código 301 deve conter um cabeçalho `Location` com a URL completa (ou seja, com descrição de protocolo e servidor) de onde o recurso está atualmente.

303 - See Other

É utilizado quando a requisição foi processada, mas o servidor não deseja enviar o resultado do processamento. Ao invés disso, o servidor envia a resposta com este código de status e o cabeçalho `Location`, informando onde a resposta do processamento está.

304 - Not Modified

É utilizado, principalmente, em requisições `GET` condicionais - quando o cliente deseja ver a resposta apenas se ela tiver sido alterada em relação a uma requisição anterior.

307 - Temporary Redirect

Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.

4xx

400 - Bad Request

É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.

401 - Unauthorized

Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação (ou a autenticação fornecida for inválida).

403 - Forbidden

Utilizado quando o cliente está tentando realizar uma operação sem ter a devida autorização.

404 - Not Found

Utilizado quando o recurso solicitado não existe.

405 - Method Not Allowed

Utilizado quando o método HTTP utilizado não é suportado pela URL. Deve incluir um cabeçalho `Allow` na resposta, contendo a listagem dos métodos suportados (separados por ",").

409 - Conflict

Utilizado quando há conflitos entre dois recursos. Comumente utilizado em resposta a criações de conteúdos que tenham restrições de dados únicos - por exemplo, criação de um usuário no sistema utilizando um *login* já existente. Se for causado pela existência de outro recurso (como no caso citado), a resposta deve conter um cabeçalho `Location`, explicitando a localização do recurso que é a fonte do conflito.

410 - Gone

Semelhante ao 404, mas indica que um recurso já existiu neste local.

412 - Precondition failed

Comumente utilizado em resposta a requisições `GET` condicionais.

415 - Unsupported Media Type

Utilizado em resposta a clientes que solicitam um tipo de dados que não é suportado - por exemplo, solicitar JSON quando o único formato de dados suportado é XML.

5xx

500 - Internal Server Error

É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica.

503 - Service Unavailable

Indica que o servidor está atendendo requisições, mas o serviço em questão não está funcionando corretamente. Pode incluir um cabeçalho `Retry-After`, dizendo ao cliente quando ele deveria tentar submeter a requisição novamente.

2.7 Conclusão

Você pôde conferir, neste capítulo, os princípios básicos do protocolo HTTP. Estes princípios são o uso de cabeçalhos, métodos, códigos de status e formatos de dados - que formam a base do uso eficiente de REST.

Você irá conferir, nos próximos capítulos, como REST se beneficia dos princípios deste protocolo de forma a tirar o máximo proveito desta técnica.

Capítulo 3

Conceitos de REST

"A mente que se abre a uma nova idéia jamais volta a seu tamanho original"
– Albert Einstein

Como dito anteriormente, REST é baseado nos conceitos do protocolo HTTP. Além disso, este protocolo é a base para a *web* como a conhecemos, sendo que a própria navegação nesta pode ser encarada como um uso de REST.

No entanto, REST não é tão simples quanto simplesmente utilizar HTTP - existem regras que devem ser seguidas para se realizar uso efetivo deste protocolo. A intenção deste capítulo é apresentar quais são estes conceitos, de maneira que você, leitor, conheça as técnicas certas para desenvolvimento deste tipo de serviço.

3.1 Semânticas de recursos

Todo serviço REST é baseado nos chamados **recursos**, que são entidades bem definidas em sistemas, que possuem identificadores e endereços (URL's) próprios. No caso da aplicação que estamos desenvolvendo, `brejaonline.com.br`, podemos

assumir que uma cerveja é um recurso. Assim, as regras de REST dizem que as cervejas devem ter uma URL própria e que esta URL deve ser significativa. Desta forma, uma boa URL para cervejas pode ser `/cervejas`.

URL's no singular ou no plural?

Não existem regras em relação a estas URL's estarem no singular ou no plural - ou seja, não importa se você prefere utilizar `/cerveja` ou `/cervejas`. O ideal, no entanto, é manter um padrão - tenha todas as suas URL's no singular ou todas no plural, mas não misturas.

De acordo com o modelo REST, esta URL realizará interação com todas as cervejas do sistema. Para tratar de cervejas específicas, são usados identificadores. Estes identificadores podem ter qualquer formato - por exemplo, suponha uma cerveja Erdinger Weissbier com código de barras 123456. Note que o nome da cerveja não serve como identificador (existem várias "instâncias" de Erdinger Weissbier), mas o código de barras, sim.

Desta forma, deve ser possível buscar esta cerveja através do código de barras, com a URL `/cervejas/123456`. Lembre-se, regras semelhantes aplicam-se a **qualquer** recurso. O identificador a ser utilizado na URL pode ser qualquer coisa que você assim desejar, e mais de um tipo de identificador pode ser usado para alcançar um recurso. Por exemplo, suponha que a sua base de clientes tenha tanto um identificador gerado pelo banco de dados quanto os CPF's dos clientes:

- Cliente número 1: CPF 445.973.986-00
- Cliente número 2: CPF 428.193.616-50
- Cliente número 3: CPF 719.760.617-92

Assim, para buscar o cliente de número 1, deve ser possível utilizar tanto a URL `/cliente/1` quanto a URL `/cliente/445.973.986-00`.

Podem haver casos, também, de identificadores compostos. Neste caso, a melhor prática seria separá-los utilizando **;** (ponto-e-vírgula). Por exemplo, um serviço de mapas em que seja possível achar um ponto por latitude e longitude teria URL's como `/local/-23.5882888;-46.6323259`.

Mais à frente, detalharei casos mais complexos de URL's.

3.2 Interação por métodos

Para interagir com as URL's, os métodos HTTP são utilizados. A regra de ouro para esta interação é que **URL's são substantivos, e métodos HTTP são verbos**. Isto quer dizer que os métodos HTTP são os responsáveis por provocar alterações nos recursos identificados pelas URL's.

Estas modificações são padronizadas, de maneira que:

- `GET` - recupera os dados identificados pela URL
- `POST` - cria um novo recurso
- `PUT` - atualiza um recurso
- `DELETE` - apaga um recurso

CRUD

Note que estes quatro métodos principais podem ser diretamente relacionados a operações de bancos de dados. Assim, para recuperar o cliente de número 1 do banco de dados, basta utilizar o método `GET` em conjunto com a URL `/cliente/1`; para criar um novo cliente, basta utilizar o método `POST` sobre a URL `/cliente` (o identificador será criado pelo banco de dados); para atualizar este cliente, utilize o método `PUT` sobre a URL `/cliente/1` e, finalmente, para apagar o cliente, utilize o método `DELETE` sobre a URL `/cliente/1`.

Note que todas estas operações são **lógicas**. Isto quer dizer que utilizar o método `DELETE`, por exemplo, não significa necessariamente excluir o dado do banco de dados - significa apenas indisponibilizar o recurso para consumo pelo método `GET`. Ou seja, `DELETE` pode apenas marcar o dado no banco de dados como **desativado**.

Tarefas

Criar serviços REST que executem tarefas de negócio (ex: enviar um e-mail, criar uma máquina em um serviço de *cloud computing*, validar um CPF e outras tarefas que não necessariamente envolvam interação com um banco de dados) é tarefa mais complexa. Exige certo grau de domínio da teoria sobre REST, e muitas vezes não existem respostas 100% assertivas em relação à corretude da solução.

Para obter resultados satisfatórios, deve-se sempre ter o pensamento voltado para a orientação a recursos. Tomemos como exemplo o caso do envio de e-mails: a

URL deve ser modelada em formato de substantivo, ou seja, apenas `/email`. Para enviar o e-mail, o cliente pode usar o método `POST` - ou seja, como se estivesse “criando” um e-mail. Da mesma forma, inicializar uma máquina em um serviço de *cloud computing* pode ter uma abordagem mais similar a um CRUD: para inicializar a máquina, o cliente utiliza o método `POST` na URL `/maquinas`. Para desligá-la, utiliza o método `DELETE`.

Existem, também, casos em que as soluções podem não parecer triviais à primeira vista. Um caso clássico é o de validações (de CPF, por exemplo). A URL pode ser claramente orientada a substantivos (`/validacao/CPF`, por exemplo), mas e quanto aos métodos?

A solução para este caso é pensar da seguinte maneira: “eu preciso *obter* uma validação? Ou criar uma, ou apagar, ou atualizar?”. Desta maneira, a resposta natural para a questão parece convergir para o método `GET`. Note que, neste caso, entra em cena o uso do método `HEAD` - poderíamos utilizar o método `GET` para conseguir resultados detalhados e o método `HEAD` para obter apenas o código de *status* da validação (lembre-se de que este método não retorna corpo!).

3.3 Representações distintas

Um dos pilares de REST é o uso de *media types* para alterar as representações de um mesmo conteúdo, sob perspectivas distintas. Esta ótica fica evidente quando se responde à pergunta: “que *lado* do recurso estou procurando eu quero enxergar? Preciso de uma foto de uma cerveja, ou da descrição dela?”. Por exemplo, ao realizar uma busca por uma cerveja no sistema, pode-se tanto desejar um XML com os dados da cerveja, quanto um JSON quanto uma foto da cerveja:

Figura 3.1: Pode ser mais interessante obter uma foto da cerveja do que a descrição dela

Com REST, tudo isso pode ser feito utilizando-se a mesma URL, por exemplo, `/cervejas/1`. O que vai modificar o resultado é a solicitação que o cliente fizer, através do cabeçalho `Accept`: se o cliente fizer uma solicitação para a URL `/cervejas/1` passando o cabeçalho `Accept` com o valor `application/xml`, ele irá obter uma representação da cerveja como XML. Mas se o cliente passar fizer esta requisição passando o valor `image/*`, ele irá obter uma foto da cerveja.

O cabeçalho Accept e os curingas

Note que, neste exemplo, eu estou falando de obter uma imagem passando o cabeçalho `Accept` com o valor `image/*`. Isto se dá devido ao fato de que, muitas vezes, não estamos interessados no tipo da imagem (JPG, GIF, PNG, etc.), mas apenas no fato de ela **ser** uma imagem!

Leonard Richardson e Sam Ruby [1] também defendem, em uma abordagem mais pragmática, que é interessante oferecer distinções entre esses tipos de dados na própria URL, a partir de extensões (similar às extensões de arquivos no sistema operacional). Por exemplo, no caso das cervejas, para obter uma representação das cervejas como XML, utilizaríamos a URL `/cervejas.xml`; para obter a representação como uma imagem jpg, utilizaríamos a URL `/cervejas.jpg`. Note que esta abordagem possui vantagens e desvantagens: ao passo que a testabilidade é simplificada (já que podemos testar os resultados pelo navegador), a implementação pode ser mais complexa e também pode induzir diferenciações nas implementações (ou seja, ter lógicas distintas para buscar dados semelhantes).

3.4 Uso correto de status codes

Algo que REST também prevê é o uso correto dos *status codes* HTTP. Na prática, isso significa conhecê-los e aplicar de acordo com a situação. Por exemplo, o código 200 (`OK`) é utilizado em grande parte das situações (o que pode acabar "viciando" o desenvolvedor), mas a criação de recursos deve retornar, quase sempre, o código 201 (`Created`) e o cabeçalho `Location`, indicando a localização do recurso criado.

Ainda, se dois clientes realizarem criação de recursos de forma que estes forneçam a chave utilizada para referenciar o recurso (ou seja, existe uma restrição de que a chave deve ser única por recurso), e estes dois clientes fornecerem a mesma chave na criação do recurso, o código de *status* 409 (`Conflict`) deve ser utilizado.

> **Posso utilizar a chave primária do meu banco de dados como ID dos recursos?**
>
> É errado assumir que a chave utilizada nos recursos será **sempre** o ID do banco de dados. É perfeitamente aceitável utilizar esta chave, mas isto deve ser uma **coincidência**, de maneira que o seu cliente **jamais** se sinta como se estivesse simplesmente utilizando uma interface diferente para acessar um banco de dados. Esta restrição é condizente com o que mencionei em meu livro SOA Aplicado: integrando com web services e além [2], de que o cliente nunca deve sentir os efeitos de uma modificação na infra-estrutura do serviço.

Por exemplo, suponha que a criação dos clientes na cervejaria utilize o CPF como chave dos recursos. Se dois clientes dos serviços tentarem se cadastrar com o mesmo CPF, um dos clientes deve receber o código 409. A resposta com esse código deve conter o cabeçalho `Location`, indicando onde está o recurso que originou o conflito.

Ao longo do livro, apresentarei situações onde o uso correto dos códigos de *status* fará a diferença no consumo dos serviços.

3.5 HATEOAS

A última das técnicas mencionadas por Roy é o uso de *Hypermedia As The Engine Of Application State* - HATEOAS. Trata-se de algo que todo desenvolvedor *web* já conhece (apenas não por esse nome).

Toda vez que acessamos uma página *web*, além do texto da página, diversos *links* para outros recursos são carregados. Estes recursos incluem *scripts* `JavaScript`, `CSS`, imagens e outros. Além disso, muitas páginas têm formulários, que encaminham dados para outras URL's.

Por exemplo, considere o seguinte trecho de página HTML:

```
<html>
   <head>
      <link rel="icon" href="/assets/favicon.ico" type="image/ico" />
      <link href="/assets/fonts.css" rel="stylesheet" type="text/css" />
   </head>
```

```
    <body>
        <img src="/assets/header.jpg" />
    </body>
</html>
```

Nela, existe uma referência para o ícone da página (referenciado com a *tag* `link` e atributo `rel="icon"`), uma referência para um arquivo CSS e uma referência para uma imagem.

HATEOAS considera estes *links* da mesma forma, através da referência a ações que podem ser tomadas a partir da entidade atual. Por exemplo, vamos modelar o sistema de compras da nossa cervejaria. Para criar um pedido de compra, é necessário fornecer um XML contendo os dados da requisição:

```
<compra>
    <item>
        <cerveja id="1">Stella Artois</cerveja>
        <quantidade>1</quantidade>
    </item>
</compra>
```

Se esta compra for aceita pelo servidor, o seguinte pode ser retornado:

```
<compra id="123">
    <item>
        <cerveja id="1">Stella Artois</cerveja>
        <quantidade>1</quantidade>
    </item>
    <link rel="pagamento" href="/pagamento/123" />
</compra>
```

Note a presença da *tag* `link`. Se o cliente utilizar a URL presente no atributo `href`, pode informar ao servidor o meio de pagamento aceito para a compra. A criação do meio de pagamento será feita através do método HTTP `POST`, conforme mencionado anteriormente. Esse conhecimento "prévio" faz parte do que se convencionou chamar, em REST, de **interface uniforme**.

Estes *links*, em HATEOAS, têm dois segmentos: *links* **estruturais** ou **transicionais**. Os links estruturais, como diz o nome, são referentes à estrutura do próprio conteúdo. Por exemplo, suponha que você busque uma listagem de endereços de um determinado cliente:

```
<cliente>
   <enderecos>
      <link href="/cliente/1/endereco/1" title="Endereço comercial" />
      <link href="/cliente/1/endereco/2" title="Endereço residencial" />
      <link href="/cliente/1/endereco/3" title="Endereço alternativo" />
   </endereco>
</cliente>
```

Estes links produzem diversos benefícios para o cliente.

O primeiro benefício, e talvez o mais óbvio, é a redução do acoplamento entre o cliente e o servidor, já que o cliente pode seguir estes links para descobrir onde estão os endereços. Caso a URL de endereços mude, o cliente não deverá sentir os impactos se seguir os *links*.

O segundo benefício é que eles exercem uma ação inteligente do ponto de vista do cliente, já que este não precisa recuperar dados de que não precisa. Se ele precisar apenas do endereço comercial, basta seguir somente o *link* que contém o endereço comercial, e não todos os outros.

O terceiro é do ponto de vista da performance. Além de recuperar um volume menor de dados, em caso de uma lista (como neste exemplo, de endereços), o cliente pode paralelizar as requisições e, assim, diminuir o tempo de obtenção das informações. Obviamente, isso depende da latência da rede (ou seja, o tempo decorrido em relação à saída da requisição da máquina cliente e chegada ao servidor, e vice-versa), mas isso pode beneficiar-se do cacheamento do recurso.

Já os links transicionais são relativos a ações, ou seja, a ações que o cliente pode efetuar utilizando aquele recurso. Você já viu este exemplo quando eu me referí ao pagamento da compra.

Existem cinco atributos importantes para os links:

- href
- rel
- title
- method
- type

Existe alguma especificação para estes links?

Existe uma especificação formal para tratamento destes links chamada XLink (disponível em http://www.w3.org/TR/xlink/) . Os atributos `href`, `title` e `method` estão definidos lá. Os outros são usados como convenção.

O atributo `href` faz referência à URL onde o recurso está localizado (apenas como lembrete, esta URL pode ser absoluta - com especificação de protocolo, *host*, porta, etc. - ou relativa -ou seja, o cliente deve buscar o recurso no mesmo servidor onde fez a primeira requisição.

O atributo `rel` é utilizado com um texto de auxílio para o uso, que não deve ser lido pelo cliente final. O valor deste atributo deve ser utilizado pelo cliente para detectar o tipo de informação presente na URL. Por exemplo, o Netflix utiliza URL's para realizar esta distinção, algo como http://schemas.netflix.com/catalog/people.directors. Caso tenha curiosidade, consulte a URL http://developer.netflix.com/docs/REST_API_Reference - onde está presente a documentação de referência para utilização da API do Netflix.

O atributo `title` contém uma descrição, legível por humanos, da informação que está presente na URL.

O atributo `method` é um dos tipos menos utilizados; quando o é, indica quais tipos de métodos HTTP são suportados pela URL (separados por vírgula).

Finalmente, o atributo `type` indica quais *media types* são suportados pela URL, e também são dos menos utilizados.

3.6 Sumário

Você conheceu, neste capítulo, as técnicas básicas de REST - ou seja, o que é necessário conhecer antes de começar a trabalhar com REST. Obviamente, muitas questões ainda não foram respondidas - como lidar com clientes concorrentes? Como trabalhar com *cache*? Como modelar casos mais avançados?

Ao longo dos próximos capítulos, você conhecerá a resposta para estas e outras questões. Vamos em frente?

Capítulo 4

Tipos de dados

"Conhecimento é poder"
– Francis Bacon

Para trabalhar de maneira eficiente com seus dados, é necessário conhecer, antes, os tipos de dados mais comuns a serem utilizados - suas vantagens, desvantagens, usos mais comuns, etc. Além disso, é necessário saber como utilizá-los na sua linguagem de programação.

4.1 XML

XML é uma sigla que significa `eXtensible Markup Language`, ou **linguagem de marcação extensível**. Por ter esta natureza extensível, conseguimos expressar grande parte de nossas informações utilizando este formato.

XML é uma linguagem bastante semelhante a HTML (`HyperText Markup Language`), porém, com suas próprias particularidades. Por exemplo, todo arquivo

XML tem **um e apenas um** elemento-raiz (assim como HTML), com a diferença de que este elemento-raiz é flexível o bastante para ter qualquer nome.

Além disso, um XML tem seções específicas para fornecimento de **instruções de processamento** - ou seja, seções que serão interpretadas por processadores de XML que, no entanto, não fazem parte dos dados. Estas seções recebem os nomes de **prólogo** (quando estão localizadas antes dos dados) e **epílogo**, quando estão localizadas depois. Por exemplo, é comum encontrar um prólogo que determina a versão do XML e o *charset* utilizado. Este prólogo tem o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Ou seja, diferente de *tags* regulares de XML (que têm o formato `<tag></tag>`), uma instrução de processamento tem o formato `<?nome-da-instrução ?>`. No exemplo, eu estou especificando a versão de XML utilizada (1.0) e o *charset* (UTF-8).

Afora esta informação, as estruturas mais básicas em um XML são as *tags* e os **atributos**, de forma que um XML simples tem o seguinte formato:

```
<?xml version="1.0" encoding="UTF-8" ?>
<tag atributo="valor">conteúdo da tag</tag>
```

Como observado no exemplo, os dados transportados pela sua aplicação podem estar presentes tanto na forma de atributos como de conteúdo das *tags*. Em um exemplo mais próximo do *case* de cervejaria, uma cerveja pode ser transmitida da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8" ?>
<cerveja id="1">
   <nome>Stella Artois</nome>
</cerveja>
```

4.2 Ferramental XML: conhecendo os XML Schemas

Por ser um formato amplamente utilizado por diversos tipos de aplicação, XML contém diversos tipos de utilitários para vários fins, a saber:

- Validação de formato e conteúdo;
- Busca de dados;
- Transformação

A ferramenta XML mais utilizada quando se trata de XML são os `XML Schemas`. Trata-se de arquivos capazes de descrever o formato quem um determinado XML deve ter (lembre-se, XML é flexível a ponto de permitir qualquer informação). Um `XML Schema`, como um todo, é análogo à definição de classes em Java: define-se os pacotes (que, em um `XML Schema`, são definidos como *namespaces*) e as classes, propriamente ditas (que, em `XML Schemas`, são os tipos).

Utiliza-se `XML Schemas` para que tanto o cliente quanto o servidor tenham um "acordo" a respeito do que enviar/receber (em termos da estrutura da informação).

Por exemplo, suponha que cada uma das cervejarias possua um ano de fundação, assim:

```
<cervejaria>
   <fundacao>1850</fundacao>
</cervejaria>
```

Se o cliente não tiver uma referência a respeito do que enviar, ele pode enviar os dados assim:

```
<cervejaria>
   <anoFundacao>1850</anoFundacao>
</cervejaria>
```

Note que, desta forma, a informação não seria inteligível do ponto de vista do cliente e/ou do serviço. Assim, para manter ambos cientes do formato a ser utilizado, pode-se utilizar um `XML Schema`.

Um `XML Schema` simples pode ser definido da seguinte forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://brejaonline.com.br/comum/v1"
    xmlns:tns="http://brejaonline.com.br/comum/v1">
</schema>
```

Um `XML Schema` é sempre definido dentro da *tag* `schema`. Esta *tag* deve conter, obrigatoriamente, a referência para o `XML Schema` http://www.w3.org/2001/XMLSchema e o atributo `targetNamespace`, que aponta qual deve ser o *namespace* utilizado pelo XML que estiver sendo validado por este `XML Schema`. Além disso, por convenção, o próprio *namespace* é referenciado no documento através da declaração `xmlns:tns`, indicando que o prefixo `tns` poderá ser utilizado no escopo deste `XML Schema`. Esta prática não é obrigatória, mas é sempre recomendada.

A estrutura dos dados, em `XML Schemas`, são definidas em termos de **elementos**. Os elementos são utilizados para definir informações a respeito das *tags*: nome da *tag*, número de repetições permitido, quais *sub-tags* são permitidas, quais atributos uma *tag* deve ter, etc.

Para definir um elemento dentro de um `XML Schema`, basta utilizar a *tag* `element`. Por exemplo, para definir uma *tag* `nome` num `XML Schema`, basta utilizar o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://brejaonline.com.br/comum/v1"
    xmlns:tns="http://brejaonline.com.br/comum/v1">
   <element name="nome" type="string" />
</schema>
```

Conforme mencionado anteriormente, a definição dos dados propriamente ditos é feita através de tipos. Estes tipos são divididos entre simples e complexos, sendo os simples aqueles que são *strings*, datas, números ou derivados destes; já os complexos, são definidos a partir da junção de elementos de tipos simples e/ou outros tipos complexos.

Tipos novos podem ser criados ou estendidos à vontade, sejam estes tipos simples ou complexos. Por exemplo, para criar um tipo simples novo, basta utilizar o seguinte:

```
<simpleType name="CEP" />
```

Neste mesmo *schema*, um tipo complexo pode ser criado da seguinte forma:

```
<complexType name="Endereco">
   <sequence>
      <element name="cep" type="tns:CEP" />
      <element name="logradouro" type="string" />
   </sequence>
</complexType>
```

Note que a definição do tipo complexo envolve a *tag* `sequence`. Esta *tag* é utilizada para determinar que os elementos nela envolvidos devem ser inseridos nesta ordem. Por exemplo, ao implementar este tipo complexo, o seguinte é aceito:

```
<endereco>
   <cep>12345-678</cep>
```

```
    <logradouro>Rua das cervejas</logradouro>
</endereco>
```

Porém, o inverso não é aceito:

```
<endereco>
    <logradouro>Rua das cervejas</logradouro>
    <cep>12345-678</cep>
</endereco>
```

Definindo formas de validação para tipos simples

Os tipos simples, como dito antes, oferecem extensibilidade em relação a outros tipos simples. Esta extensão pode ser feita para os fins mais diversos, sendo que o uso mais comum desta facilidade é para prover restrições sobre estes tipos simples. Por exemplo, no caso do CEP, sabemos que este segue uma estrutura bem clara de validação, que é o padrão `cinco dígitos (traço) três dígitos`. Ou seja, uma expressão regular pode ser utilizada para esta validação.

Esta restrição é representada utilizando-se a *tag* `restriction`. Esta *tag*, por sua vez, possui um atributo `base`, que é utilizado para indicar qual será o "tipo pai" a ser utilizado por este dado. Por exemplo, no caso de um CEP, o tipo a ser utilizado como tipo pai será `string`. Assim, o elemento `CEP` pode ser representado da seguinte forma:

```
<simpleType name="CEP">
    <restriction base="string">
    </restriction>
</simpleType>
```

Finalmente, dentro da *tag* `restriction` é que podemos definir o tipo de restrição que será aplicada ao utilizar este tipo. Podemos definir uma série de restrições, como enumerações de dados (ou seja, só é possível utilizar os valores especificados), comprimento máximo da string, valores máximos e mínimos que números podem ter, etc. Como queremos utilizar uma expressão regular, utilizamos a *tag* `pattern`, que define um atributo `value`. Através deste atributo, especificamos a expressão regular.

Assim sendo, definimos o nosso tipo CEP da seguinte forma:

```
<simpleType name="CEP">
    <restriction base="string">
```

```
      <pattern value="\d{5}-\d{3}" />
   </restriction>
</simpleType>
```

Expressões regulares

Expressões regulares são utilizadas para determinar formatos que certas strings devem ter. Por exemplo, estas expressões podem ser utilizadas para validar endereços de e-mail, números de cartão de crédito, datas, etc. Por ser um assunto demasiado comprido (que rendem livros apenas sobre isso), me limito aqui a apenas explicar o significado do padrão acima.

O símbolo `\d` significa um dígito (qualquer um). Ao ter o sinal `{5}` anexado, indica que cinco dígitos são aceitos. O traço representa a sí próprio.

Explorando tipos complexos

Como apresentado anteriormente, um tipo complexo, em um `XML Schema`, é semelhante a uma classe Java. Por exemplo, a definição de um endereço pode ser feita da seguinte forma:

```
<complexType name="Endereco">
   <sequence>
      <element name="CEP" type="tns:CEP" />
      <element name="logradouro" type="string" />
   </sequence>
</complexType>
```

Note a referência ao tipo "CEP", definido anteriormente. O prefixo `tns`, conforme mencionado, faz referência ao *namespace* do próprio arquivo (semelhante a uma referência a um pacote Java - que, no entanto, possui uma forma abreviada, que é o prefixo).

Os tipos complexos comportam diversas facilidades, como herança e definição de atributos. Por exemplo, considere o tipo complexo `Pessoa`:

```
<complexType name="Pessoa">
</complexType>
```

Supondo que uma Pessoa seja uma super definição para uma pessoa física ou jurídica, queremos que o tipo Pessoa seja abstrato. Assim, podemos definir o atributo abstract:

```
<complexType name="Pessoa" abstract="true">
</complexType>
```

Note que, desta forma, não haverá uma implementação do que for definido neste tipo, apenas de seus subtipos. Para criar uma extensão deste tipo, utiliza-se as *tags* complexContent e extension, assim:

```
<complexType name="PessoaFisica">
   <complexContent>
      <extension base="tns:Pessoa">

      </extension>
   </complexContent>
</complexType>
```

Assim, é possível definir elementos tanto no super tipo quanto no subtipo. Por exemplo, considere o seguinte:

```
<complexType name="Pessoa" abstract="true">
   <sequence>
      <element name="nome" type="string" />
   </sequence>
</complexType>

<complexType name="PessoaFisica">
   <complexContent>
      <extension base="tns:Pessoa">
         <sequence>
            <element name="cpf" type="tns:CPF" />
         </sequence>
      </extension>
   </complexContent>
</complexType>
```

Assim, a implementação deste tipo pode ficar assim:

```
<pessoaFisica>
   <nome>Alexandre</nome>
```

```
    <cpf>123.456.789-09</cpf>
</pessoaFisica>
```

Além disso, é possível definir atributos nos tipos complexos. Por exemplo, considere o seguinte:

```
<complexType name="Pessoa" abstract="true">
    <attribute name="id" type="long" />
</complexType>
```

Assim, a implementação deste tipo ficaria assim:

```
<pessoaFisica id="1" />
```

Um `XML Schema` também pode importar outros, notavelmente para realizar composições entre vários tipos diferentes. Por exemplo, considere as duas definições de `XML Schemas`:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/endereco/v1"
    elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://brejaonline.com.br/endereco/v1">

    <simpleType name="CEP">
        <restriction base="string">
            <pattern value="\d{5}-\d{3}" />
        </restriction>
    </simpleType>

    <complexType name="Endereco">
        <sequence>
            <element name="cep" type="tns:CEP" />
            <element name="logradouro" type="string" />
        </sequence>
    </complexType>
</schema>
```

Note que este `XML Schema` possui o `targetNamespace` definido como http://brejaonline.com.br/endereco/v1 .

Agora, considere um segundo `XML Schema`:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/pessoa/v1"
   elementFormDefault="qualified" xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://brejaonline.com.br/pessoa/v1">

   <simpleType name="CPF">
      <restriction base="string">
         <pattern value="\d{3}\.\d{3}\.\d{3}-\d{2}" />
      </restriction>
   </simpleType>

   <complexType name="Pessoa" abstract="true">
      <sequence>
         <element name="nome" type="string" />
      </sequence>
      <attribute name="id" type="long" />
   </complexType>

   <complexType name="PessoaFisica">
      <complexContent>
         <extension base="tns:Pessoa">
            <sequence>
               <element name="cpf" type="tns:CPF" />
            </sequence>
         </extension>
      </complexContent>
   </complexType>
</schema>
```

Note que, neste segundo `XML Schema`, o *namespace* é http://brejaonline.com.br/pessoa/v1 (ou seja, diferente do *namespace* de endereços).

Para utilizar o `XML Schema` de endereços no `XML Schema` de pessoas, é necessário efetuar dois passos: o primeiro, é definir um prefixo para o *namespace* de endereços. Por exemplo, para definir o prefixo `end`, utiliza-se o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/pessoa/v1"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://brejaonline.com.br/pessoa/v1"
   xmlns:end="http://brejaonline.com.br/endereco/v1">
```

```
   <!-- restante -->
</schema>
```

O próximo passo é informar ao mecanismo a localização do outro `XML Schema`, através da *tag* `import`:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://brejaonline.com.br/pessoa/v1"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://brejaonline.com.br/pessoa/v1"
   xmlns:end="http://brejaonline.com.br/endereco/v1">

   <import namespace="http://brejaonline.com.br/endereco/v1"
      schemaLocation="Endereco.xsd" />

   <!-- restante -->
</schema>
```

Assim, para utilizar os tipos definidos no novo arquivo, basta utilizar o prefixo `end`:

```
<complexType name="Pessoa">
   <sequence>
      <element name="endereco" type="end:Endereco" />
   </sequence>
</complexType>
```

Os tipos complexos também podem definir o número de ocorrências de seus elementos. Por exemplo, para determinar que um elemento tem número mínimo de ocorrências zero (ou seja, é opcional), utiliza-se a `tag minOccurs`:

```
<complexType name="Pessoa">
   <sequence>
      <element name="endereco" type="end:Endereco" minOccurs="0"/>
   </sequence>
</complexType>
```

Da mesma forma, é possível utilizar a `tag maxOccurs` para determinar o número máximo de ocorrências (ou seja, que um dado elemento representa uma lista). O número máximo de ocorrências pode ser delimitado a partir de um número fixo ou, caso não haja um limite definido, utiliza-se o valor `unbounded`. Assim, para determinar que uma pessoa possui vários endereços, pode-se utilizar o seguinte:

```
<complexType name="Pessoa">
   <sequence>
      <element name="endereco" type="end:Endereco" maxOccurs="unbounded"/>
   </sequence>
</complexType>
```

Isso provoca o seguinte resultado:

```
<pessoaFisica>
   <endereco>
      <cep>12345-678</cep>
      <logradouro>Rua Um</logradouro>
   </endereco>
   <endereco>
      <cep>87654-321</cep>
      <logradouro>Rua Dois</logradouro>
   </endereco>
</pessoaFisica>
```

A este ponto, você deve ter notado que o elemento raiz `pessoaFisica` não foi especificado. O elemento raiz é definido fora de tipos em um `XML Schema`, utilizando a *tag* `element`. Desta forma, é possível ter o seguinte:

```
<complexType name="PessoaFisica">
   <!-- definição de pessoa física -->
</complexType>

<element name="pessoaFisica" type="tns:PessoaFisica" />
```

4.3 Trabalhando com XML utilizando JAXB

A API (*Application Programmer Interface*) padrão para trabalhar com XML, em Java, é o JAXB (*Java Architecture for XML Binding*). Esta API trabalha essencialmente com anotações sobre classes Java, que dão instruções a respeito de como converter os dados em XML através da geração de `XML Schemas`.

Para transformar um `XML Schema` em classes Java compatíveis com JAXB, existe um utilitário presente na JDK chamado `xjc`. Supondo que seus `XML Schemas` estejam estruturados com o Maven, ou seja, presentes em uma pasta `src/main/resources`, é possível colocar estas classes na pasta certa usando o argumento `-d`:

```
xjc -d ../java Pessoa.xsd
```

O que deve produzir uma estrutura como a seguinte:

```
src/main/java
  br.com.brejaonline.endereco.v1
    Endereco.java
    ObjectFactory.java
    package-info.java
  br.com.brejaonline.pessoa.v1
    ObjectFactory.java
    package-info.java
    Pessoa.java
    PessoaFisica.java
src/main/resources
  Endereco.xsd
  Pessoa.xsd
```

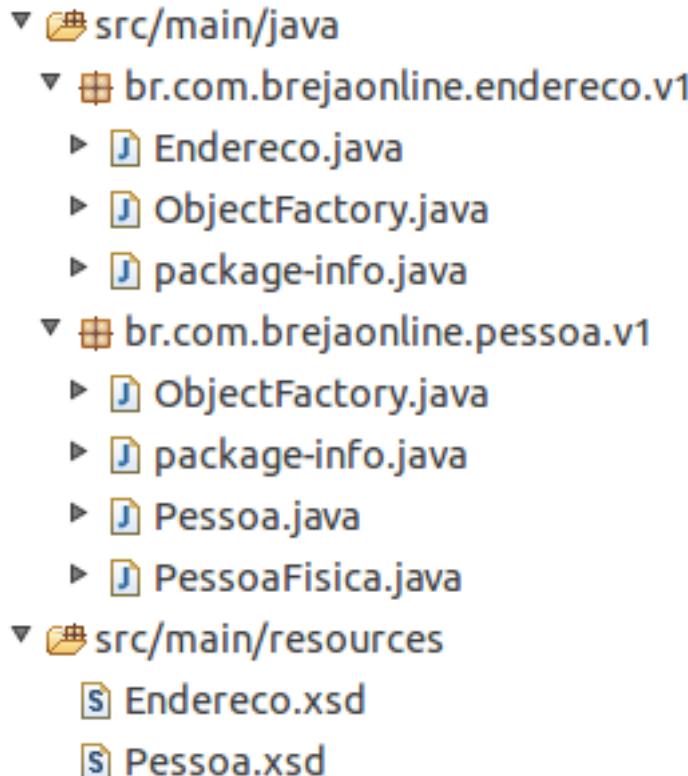



Figura 4.1: Arquivos gerados pelo xjc

Como você pode observar, uma classe para cada tipo complexo presente nos `XML Schemas` foi gerada, além das classes `ObjectFactory` e o arquivo `package-info.java`. Eles atuam, respectivamente, como uma classe fábrica para objetos recém-criados (ou seja, será utilizado no momento da tradução XML->Java) e como um mecanismo de fornecimento de informações válidas para todo o pacote.

O ARQUIVO PACKAGE-INFO.JAVA

Este arquivo é padronizado pela especificação Java (ou seja, não é definido pela especificação JAXB). Ele contém apenas a declaração de nome do pacote, com possíveis anotações. No caso do JAXB, este arquivo é gerado com a anotação `@XMLSchema`, que traz informações que serão válidas para todas as classes do pacote, como o *namespace* que será aplicado.

Vamos analisar as classes geradas:

A classe PessoaFisica

A classe `PessoaFisica` deve ter sido gerada com o seguinte código:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "PessoaFisica", propOrder = {
    "cpf"
})
public class PessoaFisica
    extends Pessoa
{

    @XmlElement(required = true)
    protected String cpf;

    // getter e setter para cpf
}
```

A anotação `@XMLAcessorType` indica para o JAXB qual o mecanismo a ser utilizado para realizar a tradução dos dados para XML. Como a *engine* do JAXB utiliza a API de *reflections* para realizar este mapeamento, o acesso aos dados pode ser feito tanto diretamente pelo valor dos atributos presentes na classe (sejam estes privados ou não) como por *getters* e *setters*.

A API disponibiliza quatro formas distintas de acesso:

- Por campos: mapeia todos os atributos de classe, independente de sua visibilidade e de estarem anotados com anotações do JAXB. Caso não estejam mapeados com nenhuma informação a respeito de nomenclatura, serão mapeados

com o nome do próprio campo. Ficam excluídos desta apenas atributos estáticos ou que sejam definidos como `transient`;

- Por propriedades: a *engine* detecta *getters* e *setters* e mapeia todos os pares encontrados;
- Por membros públicos: mapeia todas os *getters* e *setters* que tenham visibilidade ajustada para `public` e também todos os atributos de classe públicos. Este é o padrão, quando nenhum dos outros tipos é definido;
- Nenhum: mapeia apenas os membros que estejam anotados com anotações do JAXB.

A anotação `XmlType` é utilizada no momento da conversão das classes em `XML Schemas`. Ela faz menção direta ao uso de tipos complexos em `XML Schemas`, e é utilizada, neste contexto, para descrever o nome do tipo complexo e a ordem das propriedades (que devem, obrigatoriamente, estar presentes na classe).

A anotação `XmlElement` é utilizada para declarar a propriedade como elemento (que é o comportamento padrão) e outras propriedades a respeito do elemento, como a obrigatoriedade de se ter conteúdo, o nome do elemento, valor padrão, etc. Neste caso, como o elemento `cpf` não teve declarado o número mínimo de ocorrências como zero, foi assumido o valor padrão, um. Assim, o JAXB entende o elemento como obrigatório.

A classe Pessoa

A classe `Pessoa` deve ter o seguinte código:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Pessoa", propOrder = {
    "nome",
    "endereco"
})
@XmlSeeAlso({
    PessoaFisica.class
})
public abstract class Pessoa {

   @XmlElement(required = true)
   protected String nome;
```

```
    protected List<Endereco> endereco;
    @XmlAttribute(name = "id")
    protected Long id;

    // getters e setters

}
```

Como você pode notar, a classe `Pessoa` é abstrata. Isto é devido ao fato de o tipo complexo ter sido declarado também como abstrato, e as duas coisas são equivalentes em cada contexto (isto é, uma classe abstrata não poder ser instanciada, em Java, e um tipo complexo não poder ser diretamente utilizado em XML).

Sendo uma classe abstrata, o JAXB precisa saber quais são suas subclasses para que a *engine* tenha condições de fazer o mapeamento XML-> Java adequadamente. Como própria linguagem Java não permite tal detecção, entra em cena a anotação `@XmlSeeAlso`, que indica para a *engine* JAXB quais classes estão relacionadas a esta (no caso, a classe `PessoaFisica`, vista anteriormente).

Outra anotação nova é `@XmlAttribute`. Esta anotação indica que o atributo da classe é mapeado como um atributo XML, e não como elemento (comportamento padrão).

Além disso, note a presença do atributo `endereco`, que foi definido como uma lista. Este comportamento é devido ao fato de que o elemento `endereco`, declarado com número mínimo de ocorrências igual a zero e sem limite superior - o que, automaticamente, caracteriza o elemento como uma lista.

Além disso, note que a classe `Endereco` está sendo utilizada, mas não declarada na anotação `@XmlSeeAlso`. Isto é devido ao fato de que, como esta classe já é referenciada pelo próprio código (ou seja, na declaração `List<Endereco>`), não é necessário realizar a declaração na anotação.

O arquivo package-info.java

O arquivo `package-info.java` deve ter sido gerado com o seguinte conteúdo:

```
@javax.xml.bind.annotation.XmlSchema
   (namespace = "http://brejaonline.com.br/pessoa/v1",
    elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package br.com.brejaonline.pessoa.v1;
```

Note que, conforme explicado anteriormente, apenas a declaração do nome do pacote é permitida neste arquivo. Assim, a anotação `@XmlSchema` contém dados que são comuns a todas as classes do pacote, ou seja, a declaração do *namespace* e o formato dos dados, que é **qualificado**.

XML qualificado versus XML não-qualificado

Quando dizemos que um XML é qualificado, nos referimos ao formato final com que ele será formado. Em um XML qualificado, todos os elementos devem mencionar a qual *namespace* pertencem, por exemplo:

```
<pes:pessoaFisica xmlns:pes="http://brejaonline.com.br/pessoa/v1">
   <pes:cpf>123.456.789-09</pes:cpf>
</pes:pessoaFisica>
```

Note a presença do prefixo `pes` no atributo `cpf`. Se o XML não fosse qualificado, ele poderia ser escrito da seguinte maneira:

```
<pes:pessoaFisica xmlns:pes="http://brejaonline.com.br/pessoa/v1">
   <cpf>123.456.789-09</cpf>
</pes:pessoaFisica>
```

Neste caso, o fato de não ser qualificado implica, automaticamente, que o *namespace* do elemento `cpf` é igual ao de `pessoaFisica`, ou seja, http://brejaonline.com.br/pessoa/v1 .

É sempre uma boa prática utilizar XML's qualificados.

4.4 Testando o XML gerado

Para testar o formato dos dados que será produzido pelo JAXB, existe uma classe utilitária chamada `javax.xml.bind.JAXB`. Ela tem métodos para realizar *marshal* (ou seja, transformação das classes Java em XML) e *unmarshal* (ou seja, transformação de XML em Java).

Alerta sobre a classe javax.xml.bind.JAXB

Como pode ser conferido na própria documentação desta classe, esta deve ser utilizada apenas para testes, não para produção.

Para realizar o teste, basta instanciar um objeto de uma classe anotada com JAXB e utilizar o método *marshal*, passando este objeto e uma *stream* como parâmetros. Por exemplo, para mostrar o resultado no console, basta utilizar o seguinte código:

```
public static void main(String[] args) {
   PessoaFisica pessoaFisica = new PessoaFisica();
   pessoaFisica.setCpf("12345678909");
   pessoaFisica.setNome("Alexandre Saudate");

   Endereco endereco = new Endereco();
   endereco.setCep("12345-678");

   pessoaFisica.getEndereco().add(endereco);

   JAXB.marshal(pessoaFisica, System.out);
}
```

De acordo com o código acima, o seguinte deve ser apresentado:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pessoaFisica xmlns:ns2="http://brejaonline.com.br/endereco/v1"
   xmlns:ns3="http://brejaonline.com.br/pessoa/v1">
   <ns3:nome>Alexandre Saudate</ns3:nome>
   <ns3:endereco>
      <ns2:cep>12345-678</ns2:cep>
   </ns3:endereco>
   <ns3:cpf>12345678909</ns3:cpf>
</pessoaFisica>
```

4.5 Utilizando JAXB sem um XML Schema

Muitas vezes, você pode considerar desnecessário utilizar um `XML Schema`, ou muito trabalhoso gerar um. Nestes casos, é possível trabalhar com JAXB sem utilizar `XML Schemas`. Para isto, basta gerar as classes a serem utilizadas normalmente e utilizar a anotação `javax.xml.bind.annotation.XmlRootElement`.

Neste caso, podemos refatorar a classe `PessoaFisica`, por exemplo, para ser a raiz, e refatoramos toda a estrutura para remover os arquivos próprios do JAXB (ou seja, as classes `ObjectFactory` e `package-info`). A estrutura, então, fica no seguinte formato:

As classes também foram refatoradas, para ficar com o seguinte formato:

```
@XmlRootElement
public class PessoaFisica
    extends Pessoa
{

   private String cpf;

   // getters e setters
}
```

A definição de `Pessoa`:

```
@XmlSeeAlso({
    PessoaFisica.class
})
public abstract class Pessoa {

   private String nome;
   private List<Endereco> endereco;
   private Long id;

   @XmlAttribute(name = "id")
   public Long getId() {
       return id;
   }

   // getters e setters

}
```

E a classe que irá modelar o endereço:

```
public class Endereco {

   private String cep;
   private String logradouro;

}
```

O resultado do teste fica semelhante ao anterior:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pessoaFisica id="1">
```

```
   <endereco>
      <cep>12345-678</cep>
   </endereco>
   <nome>Alexandre Saudate</nome>
   <cpf>12345678909</cpf>
</pessoaFisica>
```

4.6 JSON

JSON é uma sigla para *JavaScript Object Notation*. É uma linguagem de marcação criada por Douglas Crockford e descrito na RFC 4627, e serve como uma contrapartida a XML. Tem por principal motivação o tamanho reduzido em relação a XML, e acaba tendo uso mais propício em cenários onde largura de banda (ou seja, quantidade de dados que pode ser transmitida em um determinado intervalo de tempo) é um recurso crítico.

Atende ao seguinte modelo:

- Um objeto contém zero ou mais membros
- Um membro contém zero ou mais pares e zero ou mais membros
- Um par contém uma chave e um valor
- Um membro também pode ser um *array*

O formato desta definição é o seguinte:

```
{"nome do objeto" : {
   "nome do par" : "valor do par"
   }
}
```

Por exemplo, a pessoa física pode ser definida da seguinte maneira:

```
{"pessoaFisica" : {
   "nome" : "Alexandre",
   "cpf" : "123.456.789-09"
   }
}
```

Caso seja uma listagem, o formato é o seguinte:

```
{"nome do objeto" : [
   {"nome do elemento" : "valor"},
   {"nome do elemento" : "valor"}
   ]
}
```

Novamente, a pessoa física pode ser definida da seguinte maneira:

```
{"pessoaFisica" : {
   "nome" : "Alexandre",
   [
      "endereco" : {
         "cep" : "12345-678",
         "logradouro" : "Rua Um"
      }
   ],
   "cpf" : "123.456.789-09"
   }
}
```

Note que as quebras de linha e os espaços não são obrigatórios - servem apenas como elementos para facilitar a visualização.

Também vale a pena destacar que a declaração do elemento raiz (como a declaração `pessoaFisica`) é estritamente opcional. Muitas *engines* são capazes de trabalhar com apenas os atributos do elemento. Por exemplo, a definição de pessoa física poderia ficar da seguinte forma:

```
{
   "nome" : "Alexandre",
   [
      {
         "cep" : "12345-678",
         "logradouro" : "Rua Um"
      }
   ]
   "cpf" : "123.456.789-09"
}
```

4.7 Trabalhando com JSON utilizando JAXB

Existem diversas implementações de *parsers* JSON para Java. Algumas delas são as seguintes:

- GSON
- Jackson
- Jettison
- XStream

A maneira de uso de cada uma tem vantagens e desvantagens. Uma vantagem bastante interessante de algumas é a capacidade de utilizar anotações JAXB como instruções para geração do JSON, gerando apenas um único esforço quando queremos trabalhar tanto com JSON quanto com XML. Para demonstrar o uso destas ferramentas, adicionei mais um campo na classe `PessoaFisica`, chamado `dadoTransiente`. Este campo foi anotado com `@XmlTransient`, indicando que este dado não deve ser apresentado quando o *parser* utilizar as anotações do JAXB, desta forma:

```
@XmlRootElement
public class PessoaFisica extends Pessoa {

   private String cpf;

   private String dadoTransiente = "dadoTransiente";

   @XmlTransient
   public String getDadoTransiente() {
      return dadoTransiente;
   }

   //getter e setters restantes
}
```

Vejamos o código com GSON. Para transformar uma instância desta classe em JSON com GSON, o código a seguir é utilizado:

```
PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
Gson gson = new Gson();
System.out.println(gson.toJson(pessoaFisica));
```

O que produz o seguinte resultado (formatado por mim para oferecer melhor legibilidade):

```
{"cpf":"123.456.789-09",
 "dadoTransiente":"dadoTransiente",
 "nome":"Alexandre",
 "endereco":[{"cep":"12345-678","logradouro":"Rua Um"}],
 "id":1
}
```

Como observado, é fácil gerar o conteúdo JSON utilizando GSON; no entanto, a informação transiente está presente no JSON gerado, ou seja, o GSON não oferece suporte às anotações JAXB.

Já o código com Jackson fica da seguinte forma:

```
PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
ObjectMapper objectMapper = new ObjectMapper();

AnnotationIntrospector annotationIntrospector = new JaxbAnnotationIntrospector();
objectMapper.setAnnotationIntrospector(annotationIntrospector);
System.out.println(objectMapper.writeValueAsString(pessoaFisica));
```

O que produz o seguinte resultado:

```
{"nome":"Alexandre",
 "endereco":[{"cep":"12345-678","logradouro":"Rua Um"}],
 "id":1,
 "cpf":"123.456.789-09"
}
```

Desta forma, a informação transiente foi detectada e retirada do JSON. No entanto, o código está demasiado complicado.

Vejamos o código necessário para avaliar esta informação com Jettison:

```
PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
JAXBContext context = JAXBContext.newInstance(PessoaFisica.class);
MappedNamespaceConvention con = new MappedNamespaceConvention();
Writer writer = new OutputStreamWriter(System.out);
XMLStreamWriter xmlStreamWriter = new MappedXMLStreamWriter(con, writer);
Marshaller marshaller = context.createMarshaller();
marshaller.marshal(pessoaFisica, xmlStreamWriter);
```

O código utilizado pelo Jettison é ainda mais complicado. No entanto, ele possui uma afinidade maior com o JAXB (como observado pelo uso das classes `JAXBContext` e `Marshaller`, que são próprias da API do JAXB). Isto é refletido no resultado da execução deste código:

```
{"pessoaFisica":
   {"@id":"1",
    "endereco":{"cep":"12345-678","logradouro":"Rua Um"},
    "nome":"Alexandre",
    "cpf":"123.456.789-09"
   }
}
```

Note que o JSON gerado é mais semelhante a um XML. Está presente um elemento-raiz, `pessoaFisica`, e o atributo `id` foi destacado através do prefixo `@`, indicando que este é equivalente a um atributo XML.

Finalmente, vejamos o código utilizado pelo XStream:

```
PessoaFisica pessoaFisica = Parser.criarPessoaFisicaTeste();
XStream xStream = new XStream(new JettisonMappedXmlDriver());
System.out.println(xStream.toXML(pessoaFisica));
```

Aqui, note o seguinte: o *driver* utilizado pelo XStream para geração do JSON é o do Jettison, ou seja, o XStream reutiliza outro *framework* para geração do JSON. O resultado, no entanto, é radicalmente diferente:

```
{"br.com.brejaonline.modelo.pessoa.PessoaFisica":
   {"nome":"Alexandre",
    "endereco":[
       {"br.com.brejaonline.modelo.pessoa.Endereco":
          {
           "cep":"12345-678","logradouro":"Rua Um"
          }
       }
   ],
   "id":1,
   "cpf":"123.456.789-09",
   "dadoTransiente":"dadoTransiente"
   }
}
```

Note que o XStream, por padrão, coloca os nomes das classes utilizadas na tradução e não respeita as anotações do JAXB. Toda esta informação pode ser controlada, mas com código e anotações próprias do XStream.

4.8 Validação de JSON com JSON Schema

Assim como em XML, JSON também possui um sistema de validação, conhecido como `JSON Schema`. Trata-se de um tipo de arquivo que, como `XML Schemas`, também possui um formato próprio para especificação de tipos JSON. Essa especificação está disponível no seu *site*,

Um `JSON Schema` possui como declaração mais elementar o **título** (descrito no `JSON Schema` como uma propriedade `title`). O título é o que denomina este arquivo de validação. Por exemplo, supondo que desejamos começar a descrever um `JSON Schema` para pessoas, o descritivo terá o seguinte formato:

```
{
   "title":"Pessoa"
}
```

Na sequência, é necessário definir, para este formato, o **tipo** (descrito no `JSON Schema` como uma propridade `type`). O tipo pode ser um dos seguintes:

- **array** - ou seja, uma lista de dados
- **boolean** - ou seja, um valor booleano (**true** ou **false**)
- **integer** - um número inteiro qualquer. Note que JSON não define um número de *bits* possível, apenas o fato de ser um número pertencente ao conjunto dos número inteiros.
- **number** - um número pertencente ao conjunto dos números reais. Segue a mesma regra de *integer*, sendo que *number* também contempla *integers*
- **null** - um valor nulo
- **object** - ou seja, um elemento que contém um conjunto de propriedades
- **string** - ou seja, uma cadeia de caracteres

Como pessoa é um objeto, podemos definir o tipo da seguinte forma:

```
{
   "title":"Pessoa",
   "type":"object"
}
```

Obviamente, um objeto precisa conter propriedades. Estas podem ser definidas a partir da proprieade `properties`. Por exemplo, se quisermos definir uma propriedade *nome* para uma pessoa, podemos utilizar o seguinte formato:

```
{
   "title":"Pessoa",
   "type":"object",
   "properties":{
      "nome": {
         "type":"string"
      }
   }
}
```

Note que, definida a propriedade *nome*, o conteúdo deste passa a ter o mesmo formato da raiz *pessoa*, ou seja, existe uma recursividade. É possível, portanto, definir um atributo `title` dentro de nome, definir nome como um objeto com subpropriedades, etc.

Assim sendo, podemos, portanto, definir um elemento `endereco` dentro de `pessoa`, que também será um objeto. Dessa forma, temos:

```
{
   "title":"Pessoa",
   "type":"object",
   "properties":{
      "nome": {
         "type":"string"
      },
      "endereco":{
         "type":"object",
         "properties":{
            "cep":{
               "type":"string"
            },
            "logradouro":{
               "type":"string"
            }
         }
      }
   }
}
```

Finalmente, vale a pena observar que, por padrão, o `JSON Schema` não te limita aos formatos pré-defidos (ao contrário do `XML Schema`, que apenas se atém ao definido). Por exemplo, de acordo com o `JSON Schema` acima, o seguinte JSON seria considerado válido:

```
{
   "nome":"Alexandre",
   "cpf":"123.456.789-09"
}
```

Note que, ainda que o campo `cpf` não esteja descrito, a presença do mesmo não invalida o JSON, tornando os campos meramente descritivos (ou seja, encarados como "esperados"). Caso este comportamento não seja desejável, é possível definir o atributo `additionalProperties` (que é do tipo booleano) como `false`, assim:

```
{
   "title":"Pessoa",
   "type":"object",
   "additionalProperties":false,
   "properties":{
      "nome": {
         "type":"string"
      },
      "endereco":{
         "type":"object",
         "additionalProperties":false,
         "properties":{
            "cep":{
               "type":"string"
            },
            "logradouro":{
               "type":"string"
            }
         }
      }
   }
}
```

Desta forma, o JSON descrito seria considerado inválido.

Para realizar os testes de validação JSON, é possível utilizar o programa disponível em https://github.com/fge/json-schema-validator, ou ainda, utilizar a

plataforma *online* deste, disponível até a data da escrita deste livro em http://json-schema-validator.herokuapp.com/.

Para gerar código a partir deste `JSON Schema`, é possível utilizar um programa disponível em https://github.com/joelittlejohn/jsonschema2pojo . Este programa está disponível, até a data de escrita deste livro, sob os formatos *online* (em http://www.jsonschema2pojo.org/) , como *plugin* Maven, como *plugin* Gradle, como tarefa Ant, como um programa de linha de comando ou com uma API Java.

Se preferir, pode configurar no Maven com a seguinte instrução:

```
<build>
   <plugins>
      <plugin>
         <groupId>com.googlecode.jsonschema2pojo</groupId>
         <artifactId>jsonschema2pojo-maven-plugin</artifactId>
         <version>0.3.7</version>
         <configuration>
            <sourceDirectory>${basedir}/src/main/resources/schema</sourceDirecto
            <outputDirectory>${basedir}/src/main/java</outputDirectory>
         </configuration>
         <executions>
            <execution>
               <goals>
                  <goal>generate</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

Além disso, também é preciso configurar dependências com os artefatos `common-lang` e `jackson-databind`, que serão utilizados nas classes geradas. O XML utilizado por mim para configuração destas dependências foi:

```
<dependency>
   <groupId>commons-lang</groupId>
   <artifactId>commons-lang</artifactId>
   <version>2.4</version>
</dependency>
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
```

```
    <artifactId>jackson-databind</artifactId>
    <version>2.0.0</version>
</dependency>
```

Desta forma, basta executar o comando `mvn generate-sources` para que as classes `Pessoa` (cujo nome é gerado a partir do nome do arquivo - no meu caso, `Pessoa.json`) e `Endereco` (nome definido a partir da propriedade `endereco`).

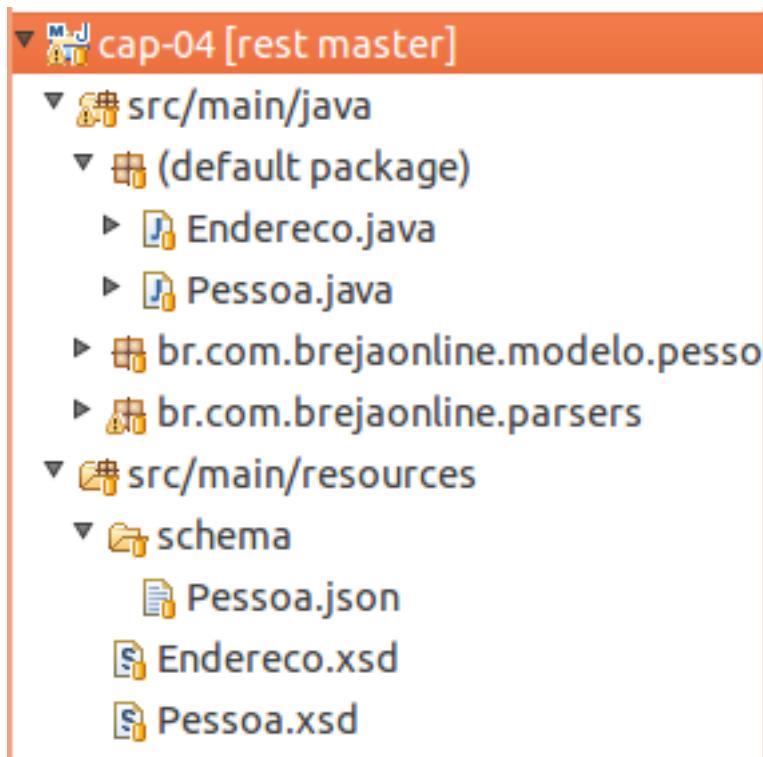



Figura 4.2: Classes geradas pelo plugin do Maven para JSON Schema

4.9 Conclusão

Nesta seção, você conheceu mais a respeito dos formatos mais utilizados em REST, ou seja, XML e JSON. Você conheceu mais a respeito do ecossistema que os cerca, ou seja, no caso do XML, `XML Schemas` e JAXB. No caso de JSON, os `JSON Schemas` e as ferramentas habilitadas a trabalharem com este formato.

Deste momento em diante, você já tem em mãos os meios que serão utilizados para construção de serviços REST, ou seja, você já conhece os princípios de REST e já sabe utilizar os tipos de dados disponíveis para utilização deste. Mas ainda falta o primordial: como combinar tudo isso?

No próximo capítulo, você verá como combinar essa técnica através da utilização de *servlets* Java. Mais adiante, você verá como utilizar API's ainda melhores. Vamos em frente?

Capítulo 5

Implementando serviços REST em Java com Servlets

"O único homem que não erra é aquele que nunca fez nada"
– Franklin Roosevelt

Agora que você já conhece os conceitos onde REST está envolvido, está na hora de ver como implementar isto em um sistema real. Como já é praxe, vou começar pela forma mais simples e, depois, partiremos para a implementação usando API's próprias para construção de serviços REST.

5.1 Uma implementação com Servlets

O leitor que já tem experiência com *servlets* Java provavelmente já imaginou, até aqui, uma implementação com esta tecnologia. Mesmo se for este o seu caso, leia esta seção até o fim - pode te trazer informações úteis mesmo a respeito do funcionamento do mecanismo da API REST de Java.

Se não for o caso, saiba que a especificação Java EE possui definida o uso de *servlets*, ou seja, trechos de código específicos para serem executados no lado do servidor. Os *servlets* são, por natureza, construídos para atenderem a qualquer protocolo de aplicação que seja transmitido pela rede; no entanto, existe uma extensão destes para trabalhar especificamente com o protocolo HTTP, que é justamente o que precisamos.

Esta extensão é a classe `javax.servlet.http.HttpServlet`, que possui os métodos conhecidos como `doXXX` (onde XXX é o nome de um método HTTP). Por exemplo, para atender a uma requisição que use o método `GET`, pode-se estender a classe `HttpServlet` e implementar o método `doGet`, assim:

```
package br.com.brejaonline.servlets;

import javax.servlet.http.*;

public class CervejaServlet extends HttpServlet {

   public void doGet (HttpServletRequest req, HttpServletResponse resp)
               throws ServletException, java.io.IOException {
      // Coloque aqui a implementação do seu código
   }
}
```

Este método vai carregar, então, a implementação do que se deseja para um serviço REST. Lembre-se de que o método `GET` é utilizado para realizar buscas no lado do servidor. Sendo este um *servlet* que busca dados de cervejas, deve-se implementá-lo de maneira a realizar esta ação.

Vamos utilizar, aqui, um conceito presente em *Domain-Driven Design*, que é o de **repositório**: uma classe de negócio especializada no armazenamento de entidades de negócio.

O que é Domain-Driven Design

Domain-Driven Design significa, numa tradução livre, *design* orientado ao domínio. Isto quer dizer, numa simplificação grosseira, que as classes que você construir devem sempre atender ao máximo possível objetivos de negócio - o que pode ser traduzido, por exemplo, em inserção de comportamento de negócio nas entidades que serão persistidas no banco de dados, o que traz ganhos em termos de coesão dos objetos. Para saber mais, sugiro ler o livro de Eric Evans, Domain-Driven Design - Atacando As Complexidades na Criação do Software.

Vamos começar, então, pela definição da entidade que desejamos buscar, ou seja, uma cerveja:

```
package br.com.brejaonline.model;

public class Cerveja {

   private String nome;
   private String descricao;
   private String cervejaria;
   private Tipo tipo;


   public enum Tipo {
      LAGER, PILSEN, PALE_ALE, INDIAN_PALE_ALE, WEIZEN;
   }
}
```

A seguir, vamos realizar a definição do nosso repositório de cervejas (ou seja, um estoque de cervejas):

```
package br.com.brejaonline.model;

import java.util.*;

public class Estoque {
   private Collection<Cerveja> cervejas = new ArrayList<>();
```

```
    public Collection<Cerveja> listarCervejas() {
        return new ArrayList<>(this.cervejas);
    }

    public void adicionarCerveja (Cerveja cerveja) {
        this.cervejas.add(cerveja);
    }
}
```

Vamos incrementar o nosso estoque para começar com algumas cervejas (afinal de contas, ninguém gosta de um estoque vazio :)). Para isso, vamos modificar a classe `Cerveja` para criar um construtor para armazenar os parâmetros e, depois, modificar o construtor de estoque para criar estas cervejas. Então, a classe `Cerveja` fica assim:

```
public class Cerveja {

    private String nome;
    private String descricao;
    private String cervejaria;
    private Tipo tipo;

    public Cerveja(String nome, String descricao, String cervejaria, Tipo tipo) {
        this.nome = nome;
        this.descricao = descricao;
        this.cervejaria = cervejaria;
        this.tipo = tipo;
    }

    // restante do código

}
```

E a classe `Estoque` fica assim:

```
public class Estoque {
    private Collection<Cerveja> cervejas = new ArrayList<>();

    public Estoque() {
        Cerveja primeiraCerveja = new Cerveja("Stella Artois",
                "A cerveja belga mais francesa do mundo :)",
```

```
            "Artois",
            Cerveja.Tipo.LAGER);
      Cerveja segundaCerveja = new Cerveja("Erdinger Weissbier",
            "Cerveja de trigo alemã",
            "Erdinger Weissbräu",
            Cerveja.Tipo.WEIZEN);
      this.cervejas.add(primeiraCerveja);
      this.cervejas.add(segundaCerveja);
   }

   //restante do código

}
```

Assim, utilizamos a classe `Estoque` no nosso *servlet* de cervejas, ficando assim:

```
public class CervejaServlet extends HttpServlet {

   private Estoque estoque = new Estoque();

   public void doGet (HttpServletRequest req, HttpServletResponse resp)
               throws ServletException, java.io.IOException {
      // Coloque aqui a implementação do seu código
   }
}
```

Mas este exemplo não segue o MVC...

Obviamente, este exemplo não é bom no quesito "acoplamento". A idéia, aqui, é apenas ser didático, e não mostrar um exemplo real. Os exemplos reais completos e refatorados estarão disponíveis alguns capítulos adiante, aguarde.

Finalmente, para facilitar a nossa vida no quesito descritividade da cerveja, vamos sobreescrever o método `toString` da cerveja:

```
public class Cerveja {
   // o código já mostrado anteriormente

   public String toString() {
```

```
        return this.nome + " - " + this.descricao;
    }
}
```

Agora, podemos utilizar este conjunto de mecanismos para listar nossas cervejas a partir do nosso *servlet*. Para isso, vamos utilizar o mecanismo de impressão dos *servlets*, que é a obtenção de um `java.io.PrintWriter` a partir da classe `javax.servlet.http.HttpServletResponse`. Essa obtenção é feita a partir do método `getWriter`:

```
package br.com.brejaonline.servlets;

import javax.servlet.http.*;
import java.io.*;
import br.com.brejaonline.model.*;

public class CervejaServlet extends HttpServlet {

    private Estoque estoque = new Estoque();

    public void doGet (HttpServletRequest req, HttpServletResponse resp)
               throws ServletException, java.io.IOException {
        PrintWriter out = resp.getWriter();
        Collection<Cerveja> cervejas = estoque.listarCervejas();
        for (Cerveja cerveja : cervejas) {
            out.print(cerveja);
        }
    }
}
```

Resta, agora, realizar o mapeamento deste *servlet* de maneira adequada. Seguindo os preceitos de REST, o ideal seria mapeá-lo para a URL `/cerveja` ou `/cervejas`, já que a nossa necessidade é buscar os dados das cervejas presentes no sistema.

Pela especificação 3.0 de *servlets*, é possível utilizar a anotação `javax.servlet.annotation.WebServlet` para descrevê-los perante o container, criando o mapeamento adequado. Portanto, nossa classe mapeada pode ficar da seguinte forma:

```
// Declaração de package e imports
```

```
@WebServlet(value = "/cervejas/*")
public class CervejaServlet extends HttpServlet {
   // código
}
```

Dica de uso

Caso tenha dificuldades em conferir o funcionamento deste código, basta ir no repositório de código-fonte do livro em https://github.com/alesaudate/rest , baixar o código e testá-lo em seu ambiente.

Finalmente, para conferir o funcionamento do código, basta abrir com o navegador o endereço http://localhost:8080/cap-05/cervejaria:

No entanto, note que algo está errado... para testar o serviço de uma maneira mais cômoda, sugiro a instalação do *plugin* `Poster`, para o navegador Mozilla Firefox, disponível em https://addons.mozilla.org/pt-br/firefox/addon/poster/ .

Uso outro navegador, o que faço?

O *plugin* também está disponível para o Google Chrome, em http://tinyurl.com/chrome-poster . Não se preocupe, o uso nos dois navegadores é realizado da mesma forma.

Uma vez instalado o *plugin* no seu Firefox, basta abrí-lo apontando para o menu Ferramentas (ou *Tools*, dependendo do idioma do seu navegador) e clicando em `Poster`:

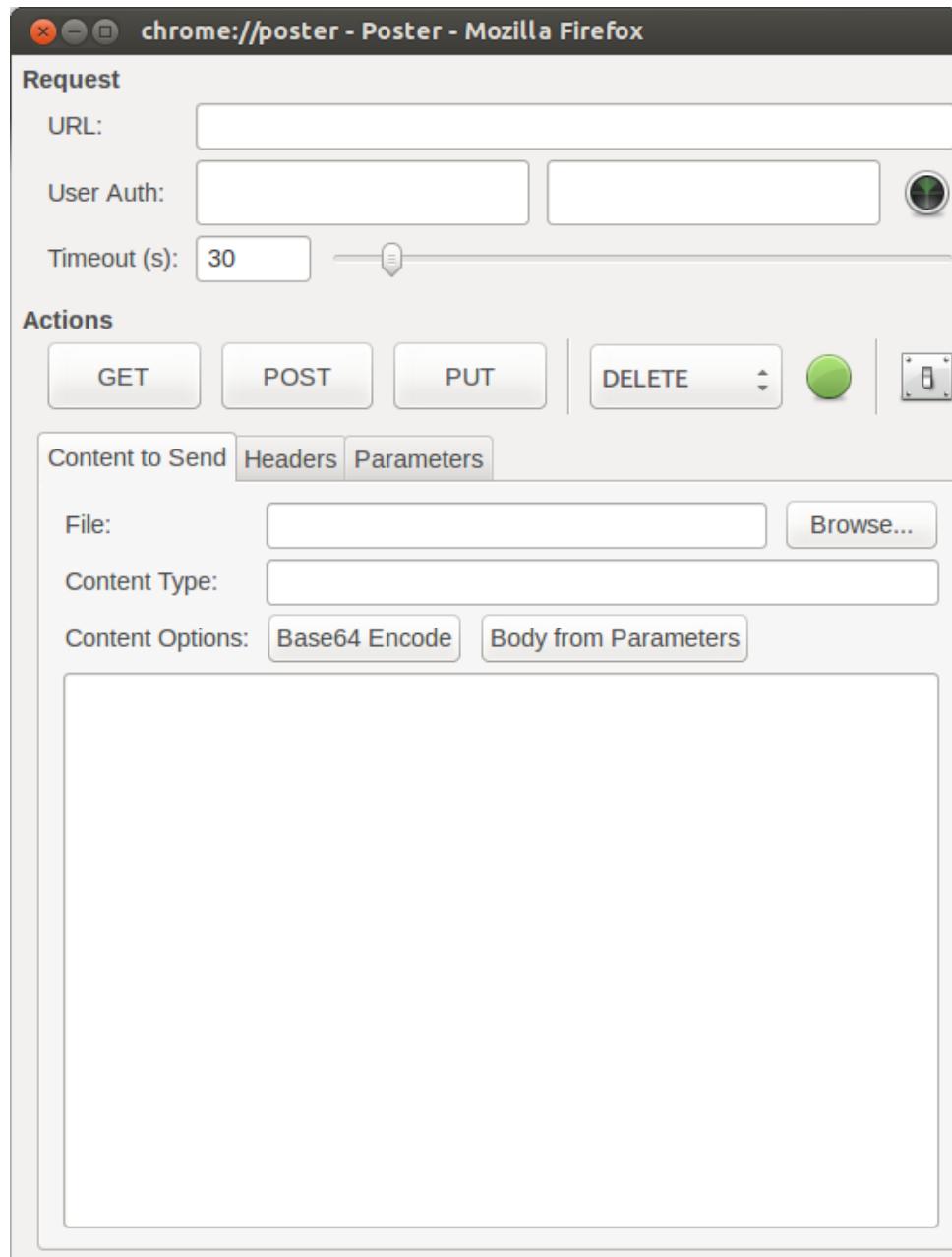



Figura 5.1: Poster, um plugin para o Firefox

Com o `Poster`, temos uma interface gráfica para testar os nossos serviços!

Na caixa `URL`, insira a URL do *servlet* de cervejas, ou seja, http://localhost:8080/cervejaria/cervejas. Na sequência, clique no botão `GET`. O resultado deve aparecer de maneira idêntica à maneira como apareceu no navegador.

Porém, algo não está certo. O cabeçalho `Content-Type` não está presente, ou seja, nem sequer existe uma definição do tipo de dados. Além disso, o resultado foi impresso como texto puro - ou seja, o cliente não terá meios para identificar os tipos de dados trafegados, etc.

Ou seja, nossa primeira missão é ajustar o tipo de dados. Para representar os dados, podemos utilizar os *MIME Types* `text/xml` ou `application/xml` para XML e `application/json` para JSON. Você viu no capítulo 4 como manipular estes tipos. Assim, suponha que o retorno desejado dessa listagem seja um conteúdo XML. Duas coisas são necessárias: criar um elemento-raiz para os elementos e realizar as adaptações para trafegar estes dados no formato desejado. Para termos o menor impacto possível, portanto, temos como opção mais fácil utilizar o JAXB: a API já vem embutida nas implementações da *Virtual Machine* Java e também oferece compatibilidade com certos mecanismos geradores de JSON.

Assim, vamos criar uma classe que encapsule as outras cervejas, denominada `Cervejas`. Esta classe será apartada do modelo, e ficará localizada no pacote `br.com.brejaonline.model.rest`. Terá o seguinte código:

```
//Declaração de pacote e imports


@XmlRootElement
public class Cervejas {

   private List<Cerveja> cervejas = new ArrayList<>();

   @XmlElement(name="cerveja")
   public List<Cerveja> getCervejas() {
      return cervejas;
   }

   public void setCervejas(List<Cerveja> cervejas) {
      this.cervejas = cervejas;
   }
}
```

Como você pode observar, é apenas uma cápsula para cervejas. Ela tem dois

motivos importantes para existir:

- Ela será usada para encapsular os outros elementos;
- Futuramente, utilizaremos essa classe para adicionar HATEOAS.

Agora, é hora de modificar a implementação do nosso *servlet* para que ele seja capaz de atender aos clientes. Para que isso aconteça, basta implementar novamente o método `doGet` no servlet, com o código de escrita de xml visto no capítulo 4:

```
//declaração de pacotes, imports e da classe

private static JAXBContext context;

static {
   try {
      context = JAXBContext.newInstance(Cervejas.class);
   } catch (JAXBException e) {
      throw new RuntimeException(e);
   }
}

protected void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {

   try {
      Marshaller marshaller = context.createMarshaller();
      resp.setContentType("application/xml;charset=UTF-8");
      PrintWriter out = resp.getWriter();

      Cervejas cervejas = new Cervejas();
      cervejas.setCervejas(new ArrayList<>(estoque.listarCervejas()));
      marshaller.marshal(cervejas, out);

   } catch (Exception e) {
      resp.sendError(500, e.getMessage());
   }
}
```

Prevenindo problemas

No código acima, note que a definição do `Content-Type` acompanha a definição do *charset*. Caso não seja definido, o container (no meu caso, o `Jetty`) normalmente seleciona um pré-definido. Observe que a ordem em que as coisas são feitas também influencia esta definição - no caso do `Jetty`, se este ajuste for feito depois da invocação a `resp.getWriter`, o container vai ignorar a definição do *charset*.

Observe que a *engine* do JAXB deposita o XML gerado diretamente na saída oferecida pela API de *servlets*. Para testar este código, recorremos novamente ao `Poster`:

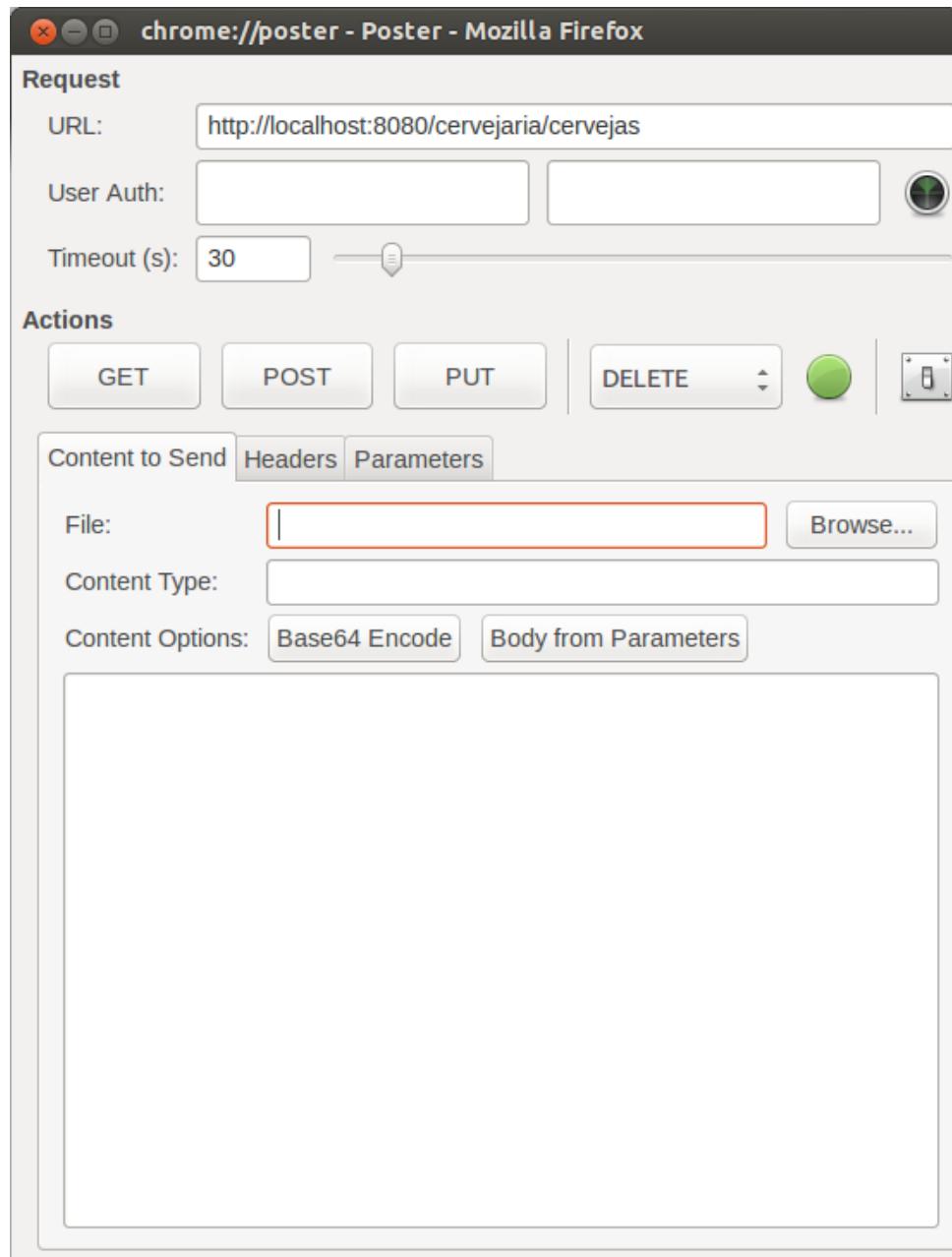
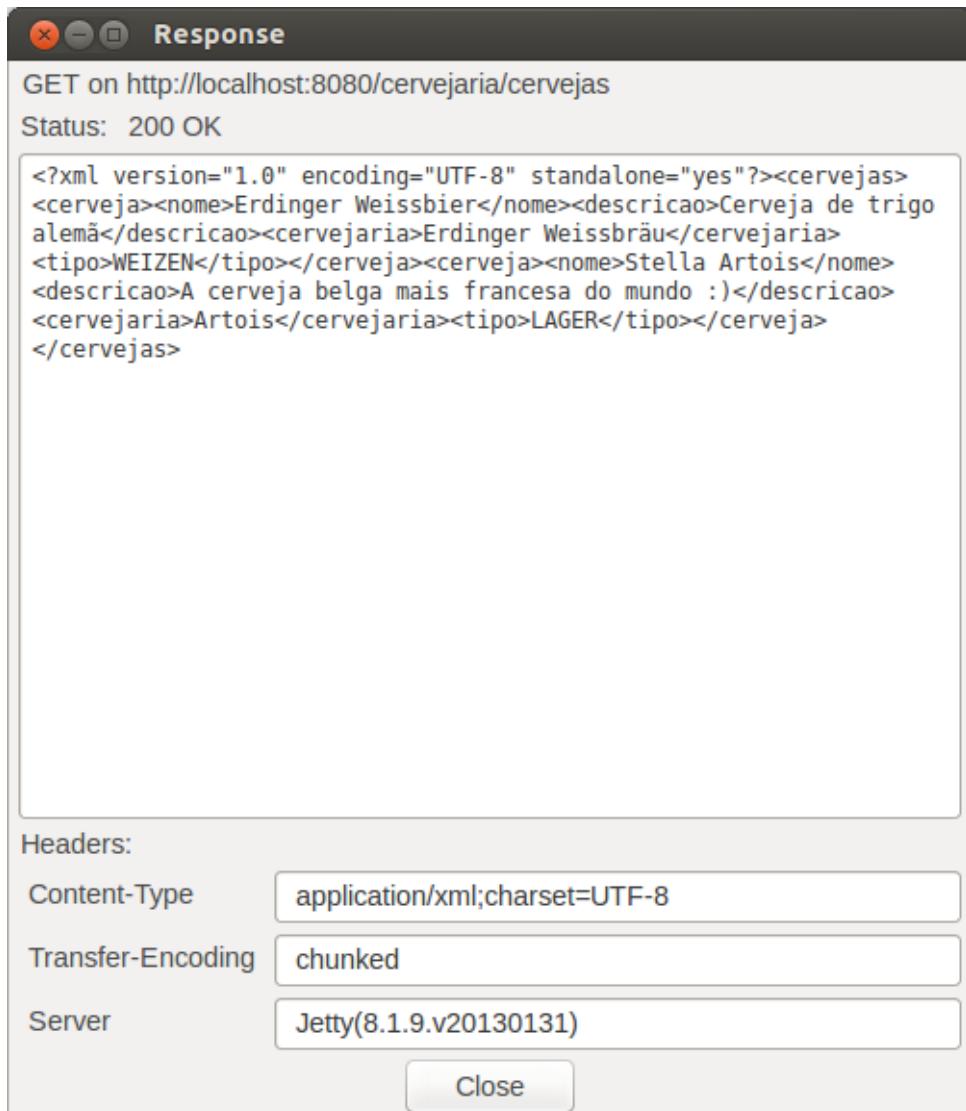



Figura 5.2: Enviando uma requisição de teste com o poster

A resposta será a seguinte:

Figura 5.3: Resposta teste poster

5.2 Implementando negociação de conteúdo

No exemplo anterior, a URL `/cervejas` está habilitada a enviar para o cliente um conteúdo apenas em formato XML. Mas o que acontece se quisermos enviar dados

de outras maneiras? Digamos que eu queira enviar dados como JSON, também, o que faço?

A saída não é criar novas URL's, mas sim, implementar uma técnica conhecida como **negociação de conteúdo**. Nesta técnica, o cliente diz para o servidor que tipo de dados deseja (através do cabeçalho `Accept`) e, então, o servidor produz o tipo de conteúdo no formato desejado.

Para implementar negociação de conteúdo com *servlets* é muito fácil; basta invocar o método `HttpServletRequest.getHeader`, assim:

```
String acceptHeader = req.getHeader("Accept");
```

A partir de então, é preciso testar o resultado. Lembre-se (de acordo com o capítulo 2) que essa negociação pode ser potencialmente compliada. Para evitar um código demasiadamente complexo, vamos apenas trabalhar com a **existência** ou não de formatos esperados neste cabeçalho. Assim, testamos primeiro para a solicitação de XML. Caso não haja XML na requisição, testamos para JSON e, se o teste falhar, devolvemos o código de erro 415 (ainda de acordo com o capítulo 2, significando que foi solicitado um formato de dados que não é suportado).

O código fica assim:

```
if (acceptHeader == null || acceptHeader.contains("application/xml")) {
    escreveXML(req, resp);
} else if (acceptHeader.contains("application/json")) {
    escreveJSON(req, resp);
} else {
    // O header accept foi recebido com um valor não suportado
    resp.sendError(415); // Formato não suportado
}
```

Agora, resta escrever os métodos `escreveXML` e `escreveJSON`. A esta altura, isto não deve ser nenhum mistério para você; mas listo abaixo caso você ainda tenha dúvidas a respeito:

```
private void escreveXML(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    Cervejas cervejas = new Cervejas();
    cervejas.setCervejas(new ArrayList<>(estoque.listarCervejas()));

    try {
```

```
        resp.setContentType("application/xml;charset=UTF-8");
        Marshaller marshaller = context.createMarshaller();
        marshaller.marshal(cervejas, resp.getWriter());

    } catch (JAXBException e) {
        resp.sendError(500); //Erro interno inesperado
    }

}

//Este código assume o Jettison como provedor de mapeamento JSON

private void escreveJSON(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    Cervejas cervejas = new Cervejas();
    cervejas.setCervejas(new ArrayList<>(estoque.listarCervejas()));

    try {
        resp.setContentType("application/json;charset=UTF-8");
        MappedNamespaceConvention con = new MappedNamespaceConvention();

        XMLStreamWriter xmlStreamWriter = new MappedXMLStreamWriter(con,
            resp.getWriter());

        Marshaller marshaller = context.createMarshaller();
        marshaller.marshal(objetoAEscrever, xmlStreamWriter);

    } catch (JAXBException e) {
        resp.sendError(500);
    }

}
```

5.3 Implementando a busca por uma cerveja específica

Até aqui, você já viu como listar todas as cervejas cadastradas no sistema. Mas como buscar uma cerveja específica?

Para resolver esta questão, o recomendado é modelar a URL de maneira que,

se o cliente fornecer o ID da cerveja na URL, apenas uma única é retornada. Por exemplo, se eu quiser buscar a Stella Artois, posso fazê-lo através da URL `/cervejas/Stella+Artois` (note que esta URL já está codificada para ser usada em um *browser*).

Para fazer isso, no entanto, é necessário realizar algumas modificações, a começar pela classe `Estoque`: esta deve ser modificada para encarar os nomes das cervejas como identificadores. Assim, substituímos a implementação da classe `Estoque` para armazenar as cervejas em um mapa, e não em uma lista:

```
private Map<String, Cerveja> cervejas = new HashMap<>();

public Collection<Cerveja> listarCervejas() {
   return new ArrayList<>(this.cervejas.values());
}

public void adicionarCerveja (Cerveja cerveja) {
   this.cervejas.put(cerveja.getNome(), cerveja);
}
```

Assim, podemos acrescentar um método nesta classe para recuperar as cervejas pelo nome:

```
public Cerveja recuperarCervejaPeloNome (String nome) {
   return this.cervejas.get(nome);
}
```

O próximo passo é incluir no *servlet* um método para extrair o identificador (ou seja, o nome da cerveja) da URL enviada pelo cliente. Para fazer isso, precisamos utilizar o método `HttpServletRequest.getRequestURI`:

```
String requestUri = req.getRequestURI();
```

De posse da URI invocada, precisamos extrair o conteúdo que está depois da última barra. Mas esta URL pode ter vários formatos:

- `/cervejas`
- `/cervejas/`
- `/cervejas/Stella`
- `/cervejas/Stella+Artois`

Ou seja, precisamos de um método que seja inteligente o suficiente para extrair o que vem depois de `/cervejas`. Existem várias maneira de fazer isso; o método criado por mim foi dividir a *string* pelos separadores (ou seja, `/`) e, depois, iterar pela lista de pedaços até detectar o contexto do *servlet*, ou seja, `cervejas`. Uma vez localizado o contexto, ajusta uma variável booleana indicando que, caso haja qualquer coisa na URL depois deste pedaço, é um identificador. O código fica assim:

```
String[] pedacosDaUri = requestUri.split("/");

boolean contextoCervejasEncontrado = false;
for (String contexto : pedacosDaUri) {
   if (contexto.equals("cervejas")) {
      contextoCervejasEncontrado = true;
      continue; //Faz o loop avançar até o próximo
   }
   if (contextoCervejasEncontrado) {
        return contexto;
   }
}
```

No entanto, este código só leva em consideração o caso do identificador existir. Para o caso da URL estar codificada, ou seja, estar como `Stella+Artois` (que, decodificado, representa `Stella Artois`), o tratamento ainda não está implementado. Mas decodificar é fácil, basta utilizar a classe `java.net.URLDecoder`:

```
if (contextoCervejasEncontrado) {
   try {
      //Tenta decodificar usando o charset UTF-8
      return URLDecoder.decode(contexto, "UTF-8");
   } catch (UnsupportedEncodingException e) {
      //Caso o charset não seja encontrado, faz o melhor esforço
      return URLDecoder.decode(contexto);
   }
}
```

Resta apenas o caso onde o identificador não existe. Para isso, o mais elegante seria modelar uma exceção própria, algo como `RecursoSemIdentificadorException`. O código finalizado fica assim:

```
private String obtemIdentificador(HttpServletRequest req)
      throws RecursoSemIdentificadorException {
```

```
    String requestUri = req.getRequestURI();

    String[] pedacosDaUri = requestUri.split("/");

    boolean contextoCervejasEncontrado = false;
    for (String contexto : pedacosDaUri) {
        if (contexto.equals("cervejas")) {
            contextoCervejasEncontrado = true;
            continue;
        }

        if (contextoCervejasEncontrado) {
            try {
                return URLDecoder.decode(contexto, "UTF-8");
            } catch (UnsupportedEncodingException e) {
                return URLDecoder.decode(contexto);
            }
        }
    }

    throw new RecursoSemIdentificadorException("Recurso sem identificador");
}
```

O próximo passo é alterar os métodos de busca para tratarem o envio tanto de uma `Cerveja` quanto da classe que pode conter todas, ou seja, `Cervejas`. Para isso, basta criar um método para realizar a localização. Como estas classes não possuem parentesco entre elas, utilizamos a classe `Object` como retorno do método:

```
private Object localizaObjetoASerEnviado(HttpServletRequest req) {
    Object objeto = null;

    try {
        String identificador = obtemIdentificador(req);
        objeto = estoque.recuperarCervejaPeloNome(identificador);
    }
    catch (RecursoSemIdentificadorException e) {
        Cervejas cervejas = new Cervejas();
        cervejas.setCervejas(new ArrayList<>(estoque.listarCervejas()));
        objeto = cervejas;
    }
```

```
    return objeto;
}
```

Finalmente, basta alterar os métodos de escrita para utilizarem este novo método. Observe que existe uma possibilidade do objeto retornado ser nulo, ou seja, caso um identificador tenha sido fornecido mas o objeto não tenha sido localizado no estoque. Assim, é necessário retornar o erro 404 (`Not Found`) caso o objeto seja nulo. O método de escrita de XML e o de JSON serão semelhantes:

```
private void escreveXML(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

    Object objetoAEscrever = localizaObjetoASerEnviado(req);

    if (objetoAEscrever == null) {
        resp.sendError(404); //objeto não encontrado
        return ;
    }

    try {
        resp.setContentType("application/xml;charset=UTF-8");
        Marshaller marshaller = context.createMarshaller();
        marshaller.marshal(objetoAEscrever, resp.getWriter());

    } catch (JAXBException e) {
        resp.sendError(500);
    }

}
```

Mas é possível enviar um Object para o JAXB?

Se você teve uma dúvida em relação ao envio da classe para o método `marshal`, você está indo no caminho certo :) . Na verdade, neste caso estamos confiando sempre que o objeto a ser enviado para este método é compatível com JAXB, e este fará a detecção em **tempo de execução**, e não em tempo de compilação. Desta forma, podemos passar qualquer objeto para o método `marshal` e, caso este não seja compatível, uma `JAXBException` será lançada.

Agora, basta testar. Use novamente o `Poster`, passando a URL http://localhost:8080/cervejaria/cervejas/Stella+Artois como parâmetro. O seguinte deverá ser retornado:

Response

GET on http://localhost:8080/cervejaria/cervejas/Stella+Artois

Status: 200 OK

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><cerveja>
<nome>Stella Artois</nome><descricao>A cerveja belga mais francesa
do mundo :)</descricao><cervejaria>Artois</cervejaria>
<tipo>LAGER</tipo></cerveja>
```

Headers:

Content-Type	application/xml;charset=UTF-8
Transfer-Encoding	chunked
Server	Jetty(8.1.9.v20130131)

Close

Figura 5.4: Retorno do teste do servlet de cervejas

5.4 Implementando a criação de um recurso

O seu *servlet* já busca recursos, mas será que está apto a realizar a criação destes?

Como você viu antes, tipicamente as criações de recursos são feitas através da utilização do método `POST`. Sendo assim, você deve apenas implementar o método `doPost` no *servlet*:

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException
```

Para criação de recursos, você tem duas opções de modelagem: fazer com que o cliente passe um identificador, já criado por ele, ou criar o identificador no lado do servidor. Como nossa aplicação é de cervejas, faz sentido deixar com que o cliente crie o identificador, já que os identificadores das cervejas são os nomes.

Como acontece em aplicações reais?

Na maior parte das aplicações REST, o servidor cria os identificadores, especialmente se o *backend* da aplicação estiver implementado com um banco de dados relacional e os ID's são gerados por este banco. Mas você deve tomar o máximo cuidado possível com esta abordagem, pois sua aplicação pode crescer e utilizar outros tipos de banco de dados. O ideal é que você sempre faça o projeto da sua API de maneira que você consiga modificar qualquer parte da sua infra-estrutura.

Desta forma, a primeira coisa que você terá que implementar é a recuperação do identificador (isto deve ser trivial, já que você já tinha feito um método para isto quando criou o método de recuperação de cervejas). Caso o identificador não seja encontrado, envie um erro 400 `Bad Request` para o cliente, indicando que a requisição deve ser alterada e re-enviada:

```
String identificador = null;
try {
   identificador = obtemIdentificador(req);
} catch (RecursoSemIdentificadorException e) {
   //Manda um erro 400
   resp.sendError(400, e.getMessage());
}
```

O próximo tratamento a ser realizado é para o caso da cerveja já existir; neste caso, um erro 409 `Conflict` deve ser lançado:

```
if (identificador != null &&
      estoque.recuperarCervejaPeloNome(identificador) != null) {

   resp.sendError(409, "Já existe uma cerveja com esse nome");
   return ;
}
```

Finalmente, você deve realizar a leitura do corpo da requisição e transformar em um objeto `Cerveja`. Para recuperar o corpo da requisição, basta executar o método `HttpServletRequest.getInputStream`. Felizmente, o restante do processo já é facilitado pelo JAXB:

```
Unmarshaller unmarshaller = context.createUnmarshaller();
Cerveja cerveja = (Cerveja)unmarshaller.unmarshal(req.getInputStream());
```

O próximo passo é garantir que a cerveja tenha o mesmo nome que o fornecido pelo identificador e, então, adicioná-la ao estoque:

```
cerveja.setNome(identificador);
estoque.adicionarCerveja(cerveja);
```

Também é necessário enviar para o cliente o código 201 `Created`, com o cabeçalho `Location`, indicando a URL onde a nova cerveja está disponível. Como nosso *servlet* já recebe a URL devidamente ajustada, basta ajustar a URL da requisição no cabeçalho `Location` através do método `HttpServletResponse.setHeader`:

```
String requestURI = req.getRequestURI();
resp.setHeader("Location", requestURI);
resp.setStatus(201);
```

Também é uma boa prática enviar de volta para o cliente o recurso criado, já que algo pode ter sido alterado na fase de criação do recurso. Este processo será bastante facilitado através do uso do método `escreveXML`, criado anteriormente:

```
escreveXML(req, resp);
```

O método completo fica assim:

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {

   try {
      String identificador = null;
      try {
         identificador = obtemIdentificador(req);
      } catch (RecursoSemIdentificadorException e) {
         resp.sendError(400, e.getMessage()); //Manda um erro 400 - Bad Request
      }

      if (identificador != null && estoque.recuperarCervejaPeloNome(identificado
         resp.sendError(409, "Já existe uma cerveja com esse nome");
         return ;
      }

      Unmarshaller unmarshaller = context.createUnmarshaller();
      Cerveja cerveja = (Cerveja)unmarshaller.unmarshal(req.getInputStream());
      cerveja.setNome(identificador);
      estoque.adicionarCerveja(cerveja);
      String requestURI = req.getRequestURI();
      resp.setHeader("Location", requestURI);
      resp.setStatus(201);
      escreveXML(req, resp);
   }
   catch (JAXBException e ) {
      resp.sendError(500, e.getMessage());
   }
}
```

Para testar este método, basta utilizar o `Poster` novamente, passando o XML da cerveja no corpo da requisição:

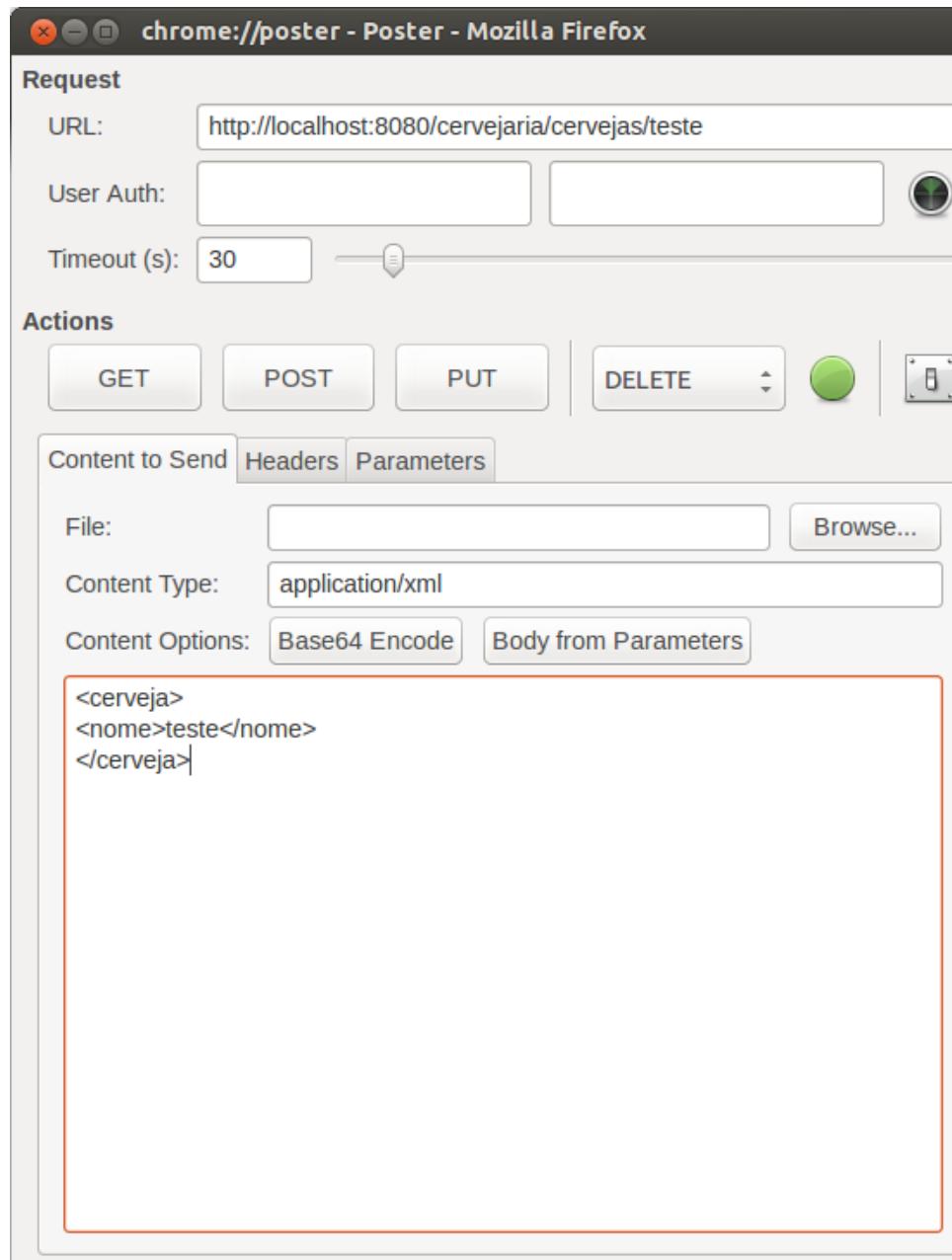



Figura 5.5: Exemplo de utilização do Poster para criação de um recurso

Caso o método tenha sido implementado corretamente, algo como o seguinte

deve ser retornado:

Response

POST on http://localhost:8080/cervejaria/cervejas/teste

Status: 201 Created

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><cerveja>
<nome>teste</nome></cerveja>
```

Headers:

Location	/cervejaria/cervejas/teste
Content-Type	application/xml;charset=UTF-8
Transfer-Encoding	chunked
Server	Jetty(8.1.9.v20130131)

Close

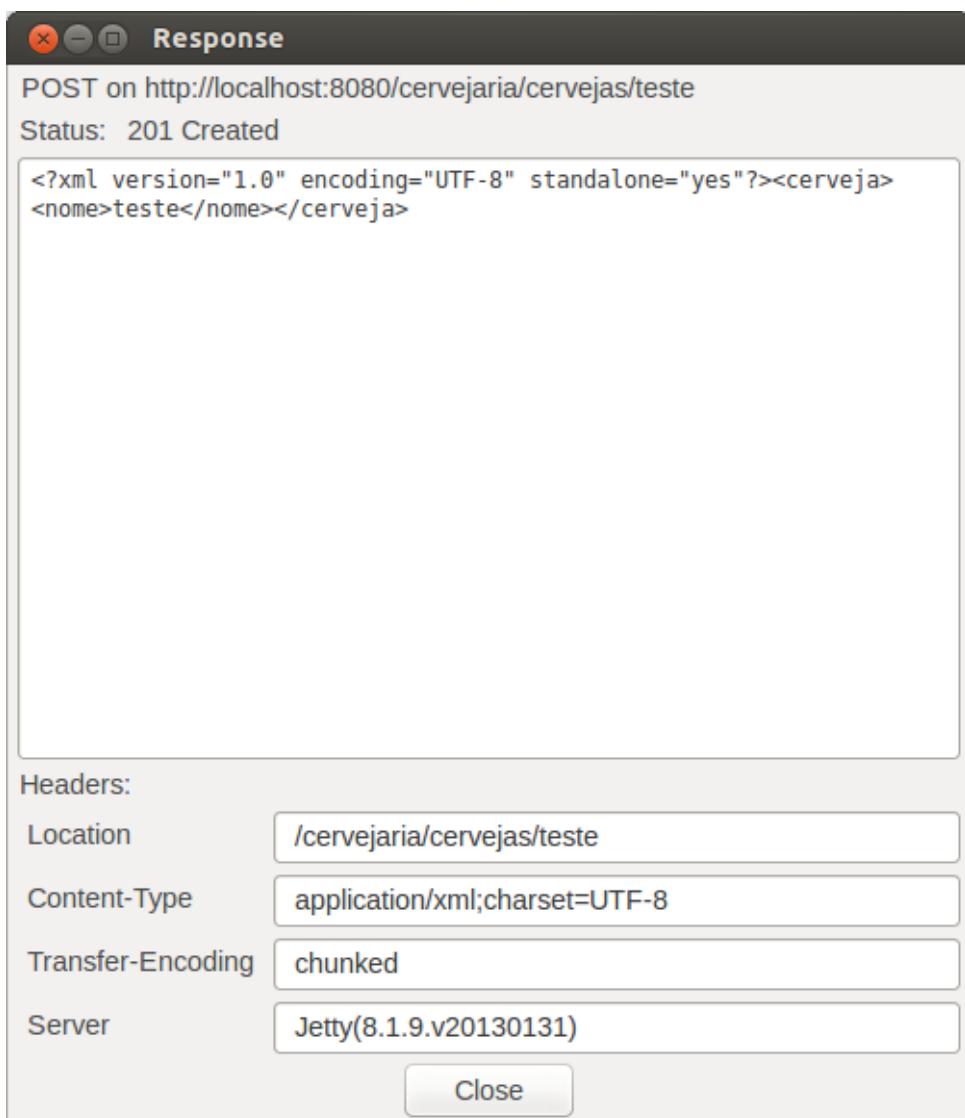



Figura 5.6: Resultado da criação do recurso

Além disso, note que, caso a mesma requisição seja submetida uma segunda vez, o código 409 deve ser retornado:

Response

POST on http://localhost:8080/cervejaria/cervejas/teste

Status: 409 J

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1"/>
<title>Error 409 Já existe uma cerveja com esse nome</title>
</head>
<body><h2>HTTP ERROR 409</h2>
<p>Problem accessing /cervejaria/cervejas/teste. Reason:
<pre>    Já existe uma cerveja com esse nome</pre></p><hr
/><i><small>Powered by Jetty://</small>
</i><br/>
<br/>
<br/>
<br/>
<br/>
<br/>
<br/>
<br/>
<br/>
<br/>
```

Headers:

Content-Type	text/html;charset=ISO-8859-1
Cache-Control	must-revalidate,no-cache,no-store
Content-Length	1442
Server	Jetty(8.1.9.v20130131)

Close

Figura 5.7: Resposta de erro indicando que o recurso já havia sido criado

5.5 Implementando negociação de conteúdo na criação do recurso

Para implementar negociação de conteúdo na criação do recurso, o cliente deve ajustar o cabeçalho `Content-Type` de maneira que indique o tipo do conteúdo presente na requisição. O servidor deve recuperar o conteúdo deste cabeçalho e tratar este conteúdo. Para fazer isso com nosso *servlet*, basta executar o método `HttpServletRequest.getContentType`:

```
String tipoDeConteudo = req.getContentType();
```

O próximo passo é um incluir um `if`/ `else if`/ `else`. O primeiro vai fazer o teste se o conteúdo é do tipo XML (`text/xml` ou `application/xml`); o segundo vai testar se o conteúdo é do tipo JSON (`application/json`) e o terceiro vai enviar um erro, informando que o tipo de dado não foi detectado.

No caso do conteúdo ser XML, o código já preparado deve ser utilizado. No caso de ser JSON, você deve utilizar um código semelhante ao de escrita. No entanto, para ler JSON, devemos realizar a leitura do corpo da requisição manualmente - o modelo do Jettison não comporta a leitura de uma *stream*. Neste caso, utilizamos a classe `org.apache.commons.io.IOUtils` para realizar a leitura das linhas de dados presentes na *stream*:

```
List<String> lines = IOUtils.readLines(req.getInputStream());
StringBuilder builder = new StringBuilder();
for (String line : lines) {
   builder.append(line);
}
```

Meu projeto não tem a biblioteca da Apache, o que faço?

Caso você utilize o Maven, basta adicionar a dependência:

```
<dependency>
   <groupId>commons-io</groupId>
   <artifactId>commons-io</artifactId>
   <version>2.4</version>
</dependency>
```

Se não for seu caso, você também pode fazer o *download* desta biblioteca no site: http://commons.apache.org/proper/commons-io/

Na sequência, basta utilizar o método de leitura do Jettison para transformar o JSON em um objeto `Cerveja`:

```
MappedNamespaceConvention con = new MappedNamespaceConvention();
JSONObject jsonObject = new JSONObject(builder.toString());

XMLStreamReader xmlStreamReader = new MappedXMLStreamReader(jsonObject, con);

Unmarshaller unmarshaller = context.createUnmarshaller();
Cerveja cerveja = (Cerveja)unmarshaller.unmarshal(xmlStreamReader);
```

A partir daqui, o método já está praticamente pronto. Basta, agora, repetir os passos realizados para leitura de XML (não esquecendo de substituir o método final por `escreveJSON`, ao invés de `escreveXML`):

```
cerveja.setNome(identificador);
estoque.adicionarCerveja(cerveja);
String requestURI = req.getRequestURI();
resp.setHeader("Location", requestURI);
resp.setStatus(201);

escreveJSON(req, resp);
```

Para testar, basta repetir o procedimento feito anteriormente, mas com uma requisição JSON:

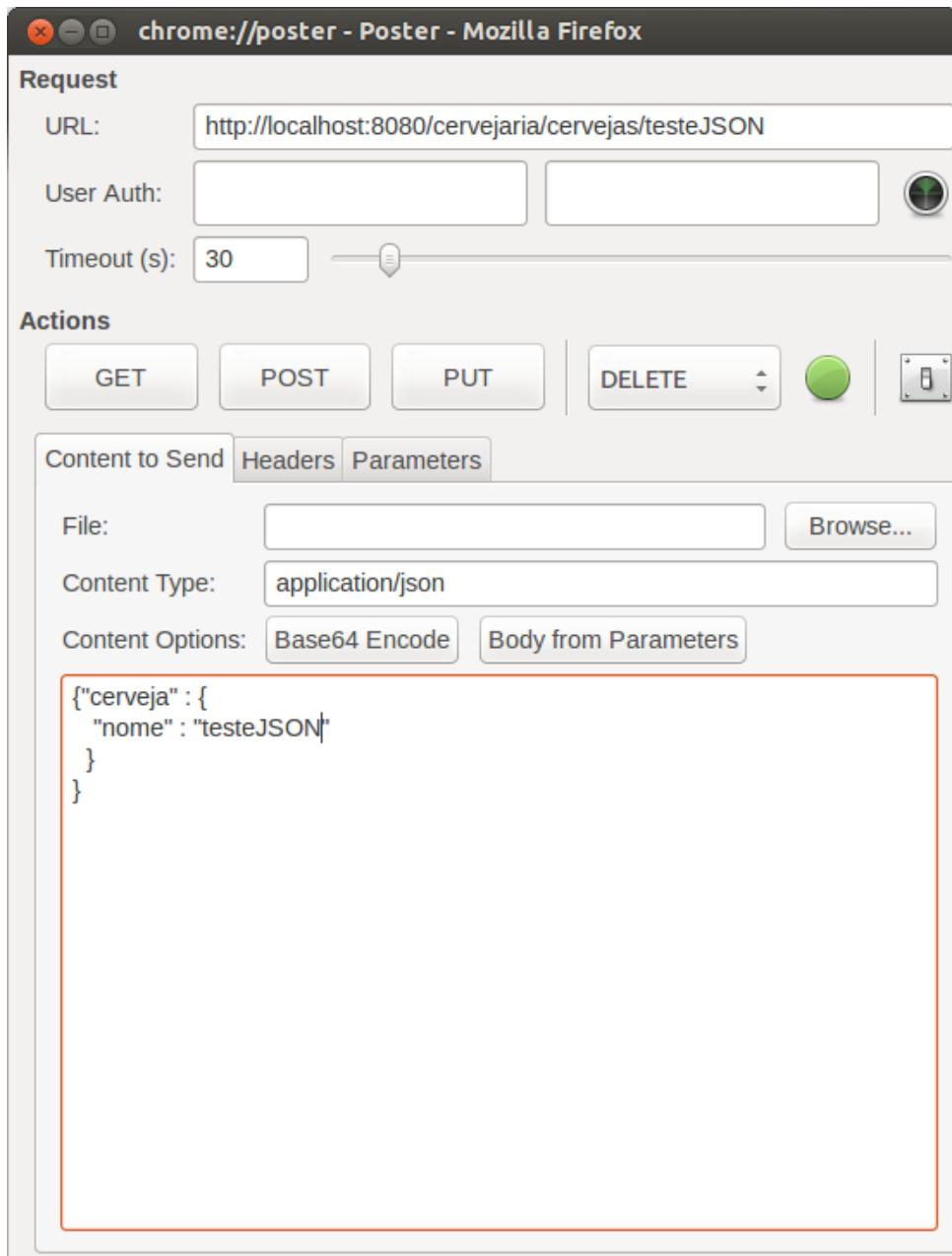



Figura 5.8: Requisição JSON

Se estiver correto, o resultado deve ser o seguinte:

Response

POST on http://localhost:8080/cervejaria/cervejas/testeJSON

Status: 201 Created

{"cerveja":{"nome":"testeJSON"}}

Headers:

Location	/cervejaria/cervejas/testeJSON
Content-Type	application/json;charset=UTF-8
Transfer-Encoding	chunked
Server	Jetty(8.1.9.v20130131)

Close

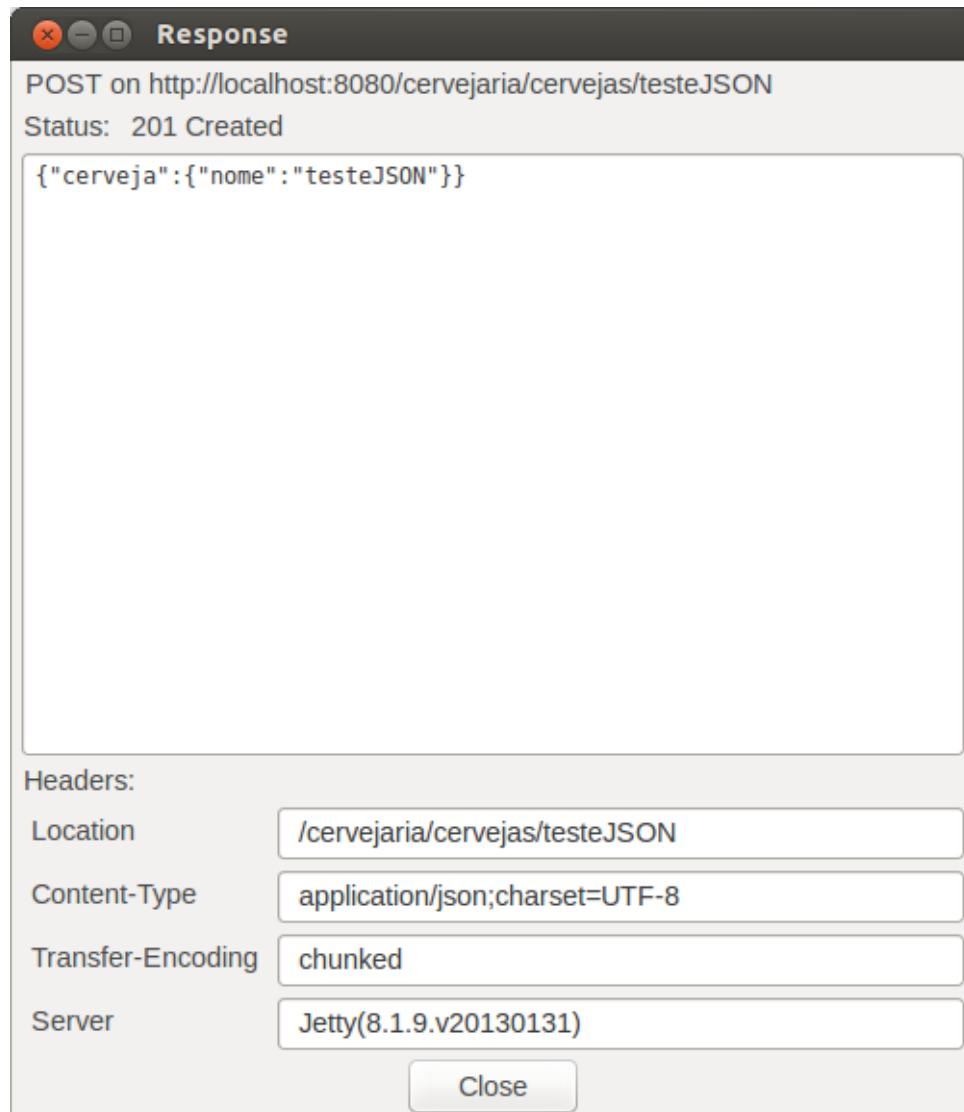



Figura 5.9: Resposta

Note que o recurso está presente no servidor independente do seu conteúdo, ou seja, se a requisição for feita uma segunda vez e não houver tipo de conteúdo definido, será retornado um XML.

5.6 Conclusão

Você viu neste capítulo como implementar serviços REST a partir de *servlets* Java. Estes serviços fazem apenas a criação e recuperação de conteúdo (o restante são apenas variações destes dois tipos).

No entanto, você deve ter notado: é demasiado complexo realizar este processo, mesmo para um recurso simples. Você deve estar se perguntando: existe alguma maneira, mais simples, de realizar este proceso? Sim, há!

Nos próximos capítulos, você verá como utilizar a especificação Java para criação de serviços REST, o JAX-RS. Além disso, você também verá como incluir *links* HATEOAS nos seus recursos de maneira eficiente, como testar esses serviços, tópicos sobre casos complexos de modelagem e muito mais. Vamos em frente?

Referências Bibliográficas

[1] Sam Ruby Leonard Richardson. Restful web services. 2007.

[2] Alexandre Saudate. Soa aplicado: integrando com web services e além. 2013.